Disk Inside

Disk Inside

source code and executables for all articles

AMIGA WORLD TECH JOURNAL

ON DISK

(See page 35.)

Animation Routines

Requesters and Gadgets

Revision Control System: A source-code manager

Plus source code and executables for more articles

U.S.A. $15.95
Canada $19.95

Volume 2, Number 2 April 1992 An IDG Communications Publication

Affordable Excellence

ReSource macro disassembler — NEW VERSION!

ReSource V5 is an intelligent interactive disassembler for the Amiga programmer. **ReSource V5** is *blindingly* fast, disassembling literally hundreds of thousands of lines per minute from executable files, binary files, disk tracks, or directly from memory. Full use is made of the Amiga windowing environment, and there are over 900 functions to make disassembling code easier and more thorough than its ever been.

Virtually all V2.0 Amiga symbol bases are available at the touch of a key. In addition, you may create your own symbol bases. Base-relative addressing, *using any address register*, is supported for disassembling compiled programs. All Amiga hunk types are supported for code scan.

ReSource V5 runs on *any* 680x0 CPU, but automatically detects the presence of an 020/030 CPU and runs *faster* routines if possible. **ReSource V5** understands 68030 instructions and supports the new M68000 Family assembly language syntax as specified by Motorola for the new addressing modes used on the 020/030 processors. **ReSource V5** and **Macro68** are among the few Amiga programs now available that provide this support. Old syntax is also supported as a user option.

An all new online help facility featuring hypertext word indexing is included. This enables you to get in-depth help about any function at the touch of a key! **ReSource V5** includes a new, completely rewritten manual featuring two tutorials on disasssembly, and comprehensive instructions for utilizing the power in **ReSource V5.**

ReSource V5 will enable you to explore the Amiga. Find out how your favorite program works. Fix bugs in executables. Examine your own compiled code.

"If you're serious about disassembling code, look no further!"

ReSource V5 requires V1.3 or later of the Amiga OS, and at least 1 megabyte of ram. **ReSource V5** supercedes all previous versions.

Suggested retail price: US$150

Buy Macro68 and ReSource together and get $30 off!

Macro68 macro assembler — NEW VERSION!

Macro68 is the *most* powerful assembler for the entire line of Amiga personal computers.

Macro68 supports the entire Motorola M68000 Family including the MC68030 and MC68040 CPUs, MC68881 and MC68882 FPUs and MC68851 MMU. The Amiga Copper is also supported, eliminating the need for tedious hand coding of 'Copper Lists'.

This fast, multi-pass assembler supports the new Motorola M68000 Family assembly language syntax, and comes with a utility to convert old-style syntax source code painlessly. The new syntax was developed by Motorola specifically to support the addressing capabilities of the new generation of CPUs. Old-style syntax is also supported, at slightly reduced assembly speeds.

Most features of **Macro68** are limited only by available memory. It also boasts macro power unparalleled in products of this class. There are many new and innovative assembler directives. For instance, a special structure offset directive assures maximum compatibility with the Amiga's interface conventions. A frame offset directive makes dealing with stack storage easy. Both forward and backward branches, as well as many other instructions, may be optimized by a sophisticated N-pass optimizer. Full listing control, including cross-referenced listings, is standard. A user-accessible file provides the ability to customize directives and run-time messages from the assembler.

Macro68 is fully re-entrant, and may be made resident. An AREXX™ interface provides "real-time" communication with the editor of your choice. A number of directives enable **Macro68** to communicate with AmigaDos™. External programs may be invoked on either pass, and the results interpreted. Possibly the most unique feature of **Macro68** is the use of a shared-library, which allows resident preassembled include files for incredibly fast assemblies.

Macro68 is compatible with the directives used by most popular assemblers. Output file formats include executable object, linkable object, binary image, and Motorola S records. **Macro68** requires at least 1 meg of memory.

Suggested retail price: US$150

The Puzzle Factory, Inc.
P.O. Box 986, Veneta, OR 97487
"Quality software tools for the Amiga"
For more information, call today! Dealer inquires invited.
Orders: (800) 828-9952
Customer Service: (503) 935-3709

Amiga and AmigaDOS are trademarks of Commodore-Amiga

VISA / MasterCard

Check or money order accepted no CODs.

Linda Barrett Laflamme, *Editor*
Louis R. Wallace, *Technical Advisor*
Janice Crotty, Vinoy Laughner, Peg LePage, *Associate Editors*

Peer Review Board

Gary Bonham, Brad Carvey, Joanne Dow, Keith Doyle, Andy Finkel, John Foust, Jim Goodnow, Eric Giguere, Scott Hood,
David Joiner, Deron Kazmaier, Sheldon Leemon, Dale Luck, Jim Mackraz, R.J. Mical, Eugene Mortimore, Bryce Nesbitt, Carolyn Scheppner,
Leo Schwab, Tony Scott, Mike Sinz, Richard Stockton, Steve Tibbett, John Toebes, Doug Walker, Dan Weiss, Mike Weiblen

Mare-Anne Jarvela, *Manager, Disk Projects*

Ann Dillon, Laura Johnson, *Designers*
Alana Korda, *Manufacturing Manager*
Debra A. Davies, *Production Supervisor*

Kenneth Blakeman, *National Advertising Sales Manager/ General Manager, TechMedia Video*
Barbara Hoy, *Sales Representative*
Michael McGoldrick, *Sales Representative*
Giorgio Saluti, *Associate Publisher, West Coast Sales*
2421 Broadway, Suite 200, Redwood City, CA 94063
415/363-5230
Heather Guinard, *Sales Representative, Partial Pages*
800/441-4403, 603/924-0100
Meredith Bickford, *Advertising Coordinator*

Wendie Haines Marro, *Marketing Director*
Laura Livingston, *Marketing Coordinator*
Margot L. Swanson, *Customer Service Representative, Advertising Assistant*
Lisa LaFleur, *Business Manager*
Mary McCole, *Publisher's Assistant*
Debbie Walsh, *Circulation Manager*

Lynn Lagasse, *Production Director*

Linda Ruth, *Single Copy Sales Director*
William M. Boyer, *Director of Credit Sales & Collections*

Dale Strang, *President/Publisher*
Susan M. Hanshaw, *Director of Operations*
Douglas Barney, *Editorial Director*

The AmigaWorld Tech Journal (ISSN 1054-4631) is an indepedent journal not connected with Commodore Business Machines, Inc. *The AmigaWorld Tech Journal* is published bi-monthly by IDG Communications/Peterborough, Inc., 80 Elm St., Peterborough, NH 03458. U.S. Subscription rate is $69.95, Canada and Mexico $79.95, Foreign Surface $89.95, Foreign Airmail $109.95. All prices are for one year. Prepayment is required in U.S. funds drawn on a U.S. bank. Application to mail at 2nd class postage rates is pending at Peterborough, NH and additional mailing offices. Phone: 603-924-0100. Entire contents copyright 1991 by IDG Communications/Peterborough, Inc. No part of this publication may be printed or otherwise reproduced without written permission from the publisher. **Postmaster:** Send address changes to *The AmigaWorld Tech Journal*, 80 Elm St., Peterborough, NH 03458. *The AmigaWorld Tech Journal* makes every effort to assure the accuracy of articles, listings and circuits published in the magazine. *The AmigaWorld Tech Journal* assumes no responsibility for damages due to errors or omissions. Third class mail enclosed.

Bulk Rate
US Postage Paid
IDG Communications/Peterborough, Inc.

MESSAGE PORT

See what's new...

First off, I should attend to some unfinished business from last issue. In it I promised we would be starting "a (currently unnamed) column to cover introductory *Amiga* programming concepts in BASIC and C." We now have a name *and* a column: Turn to page 14, and you'll find the first installment of "Beginner's Connection." Each issue, veteran Commodore author Jim Butterfield will introduce you to new system routines and Amiga programming issues, showing you how to access these functions from both Amiga Basic and C. We hope this helps not only programmers new to the Amiga, but also BASIC programmers who want to make the switch to C.

As we sorted through the letters and e-mail trying to find the best focus for "Beginner's Connection," we discovered another neglected but rather vocal group—authoring system programmers. In a flurry of holiday goodwill, we instituted yet another new column—"Authoring Solutions." Over the course of the year, we will cover all of the major authoring systems—AmigaVision, CanDo, Foundation, and VIVA—concentrating on only one per column. Each issue, an expert on the highlighted system will explain under- or undocumented features and help you improve your scripts. We kick off the column (on page 24) with AmigaVision tips from John Gerlach of IMSATT, AmigaVision's creators.

All the talk and excitement about authoring systems lately makes me wonder where programming is going:

• Will the "anybody can do it" style of high-level, icon-based authoring move in from the fringe to relegate low-level languages such as C and assembly to "for traditionalists only" status?

• Should the *Tech Journal*, like Commodore, embrace AmigaVision rather than Amiga Basic as the tool of choice for beginners?

• Should we have started only "Authoring Solutions" instead of two new columns?

For now I think the answer is no to all three questions, which I suppose marks me as command-based traditionalist instead of an iconified free thinker.

How do you feel on the authoring question—are icon-based languages the coming standard or just a shortcut for people who don't want to learn "real" programming? Drop me a line (*The AmigaWorld Tech Journal*, 80 Elm St., Peterborough, NH 03458, or llaflamme on BIX) and let me know. You ask for it, and we'll cover it. ■

[illegible]

Under the Lid: The Amiga Custom Chip Set

What do Agnus, Denise, and Paula really do all day?

By Dave Haynie

WHEN THE AMIGA 1000 was introduced, it was hailed as a true innovation in microcomputers. The WIMP interface, multitasking, and 68000 processor were all state of the art, but the main reason for excitement was the system's sophisticated set of three custom chips—Agnus, Denise, and Paula. After several generations of Amigas, the names of the custom chips have become household names, but the details of their operation are still in need of explanation.

MODERN ARCHITECTURE

Most computers have a variety of basic needs: video display, sound, floppy disk, serial port, and so on. These needs are often met with a variety of specific purpose parts, as in the run of the mill IBM clone. While that approach does work, the result is often more expensive with a much less cohesive architecture than a machine designed from the ground up to act as a system. The Amiga custom chips represent such an integrated-support chip design.

The Amiga chip set's rather novel architecture integrates the basics of video display, sound generation, floppy-disk I/O, serial-port I/O, and a few other system-support functions into three full custom chips. Based on the gate density available using full custom LSI (even back in 1985), however, most of these items are enhanced beyond the equivalent function in traditional computers. For example, a bit-image manipulation device (the Blitter) and display-list coprocessor (the Copper) were added to support the basic video display. Also, an eight-channel sprite engine makes the display of the pointer and other small moving icons much less CPU intensive. Floppy-disk I/O is driven by a DMA channel, which is very important in a multitasking environment. Similarly, the sound is based on digital-to-analog conversion, rather than the simple multiple-voice ADSR chips typical of personal computers with integrated sound. Plus, the chip set supports four of these audio channels, each with an attached DMA channel that make them as easy to use as classic "sound chips." Along with all this comes a "free" DRAM controller.

THE CHIP BUS

To understand the specifics of the novel architecture of the three custom chips, which essentially function as one, you should start with the basic bus design of the Amiga chips. Figure 1 is a very simplified block diagram of both the chip-interconnect and internal chip functions. Considered a computer system unto themselves, the three chips function like a traditional microprocessor system. The core of this system is the address-generator chip, Agnus. (The other two are the display-generator chip, Denise, and the ports and audio chip, Paula.) Like a microprocessor, Agnus is responsible for generating addresses on the address bus, managing data on a data bus, and controlling additional bus management signals. Agnus connects directly to a 16-bit chip-data bus and a multiplexed (to allow direct connection to DRAM) chip-address bus up to ten bits wide. The chip bus is a pure synchronous, deterministic bus; there are no start or stop strobes, no wait states, and thus no need for any real bus-control lines generated by Agnus.

The Agnus chip, however, is not simply addressing memory with each bus cycle. Unlike a traditional microprocessor, which must manage a small number of general purpose registers, Agnus manages many special-purpose registers or entities represented as registers, which can be located in one or both of the other custom chips. So along with a RAM address, each cycle gets a "register" address, on a separate eight-bit-wide bus called the register address, or RGA, bus. The value on the RGA bus usually indicates the access of a particular chip-bus register, although in some cases the value actually represents a synchronization strobe, DMA channel FIFO access, or miscellaneous command, such as memory refresh or NOP. Because all Amiga registers are either read-only or write-only, the RGA address also implies the direction of data flow, which is generally between an RGA-defined register and DRAM, or DRAM and a register location.

CHIP-BUS DMA

One of the highly touted features of the Amiga's custom chip system is its multiple DMA channels. Not only are there a large number of DMA channels, but DMA on the Amiga chip bus works quite differently than traditional microprocessor DMA. In most microprocessor systems, high-bandwidth I/O devices may be married to DMA controllers of some form. A DMA controller makes I/O data transfers more efficient, because it can handle the jobs of I/O polling and data transfer to memory, freeing up the host CPU for other activity. Generally, such a DMA controller queues up a reasonable amount of data in a FIFO, requests the microprocessor bus, receives a bus grant, transfers all data as bus master, then relinquishes the bus.

While that system works just fine in the nondeterministic world of the traditional microprocessor, it can't work on the Amiga chip bus. Amiga DMA channels manage such processes as video display, audio playback, and disk I/O. If any of these does not happen exactly when it's supposed to, the result is a failure of some kind. So on the Amiga chip bus, much of the DMA is not truly arbitrated. It's preallocated to "slots" based on the video raster, as illustrated in Figure 2. Agnus simply generates the proper RGA command at the appropriate slot, and the correct DMA channel is accessed. Not only does this allow the chip bus to be totally deterministic, but also it keeps all address generation in Agnus, simplifying the design of the other chips on the bus.

Each slot indicated is approximately 560ns long and consists of two possible chip bus cycles. The first half of the slot, the "even" half, is usually devoted to a fixed allocation, such as refresh or audio DMA. Fixed DMA only takes place if it's enabled. For example, sprites that aren't turned on don't get cycles, and video fetch uses only the cycles its display mode requires. Video and sprites can actually overlap, as sprites are subtracted to permit overscan video fetch. Any free cycles are made available to nondeterministic chip resources. The first of these to get cycles is the Copper, then the Blitter, and finally, the host-processor access gate (more on that later), based on need.

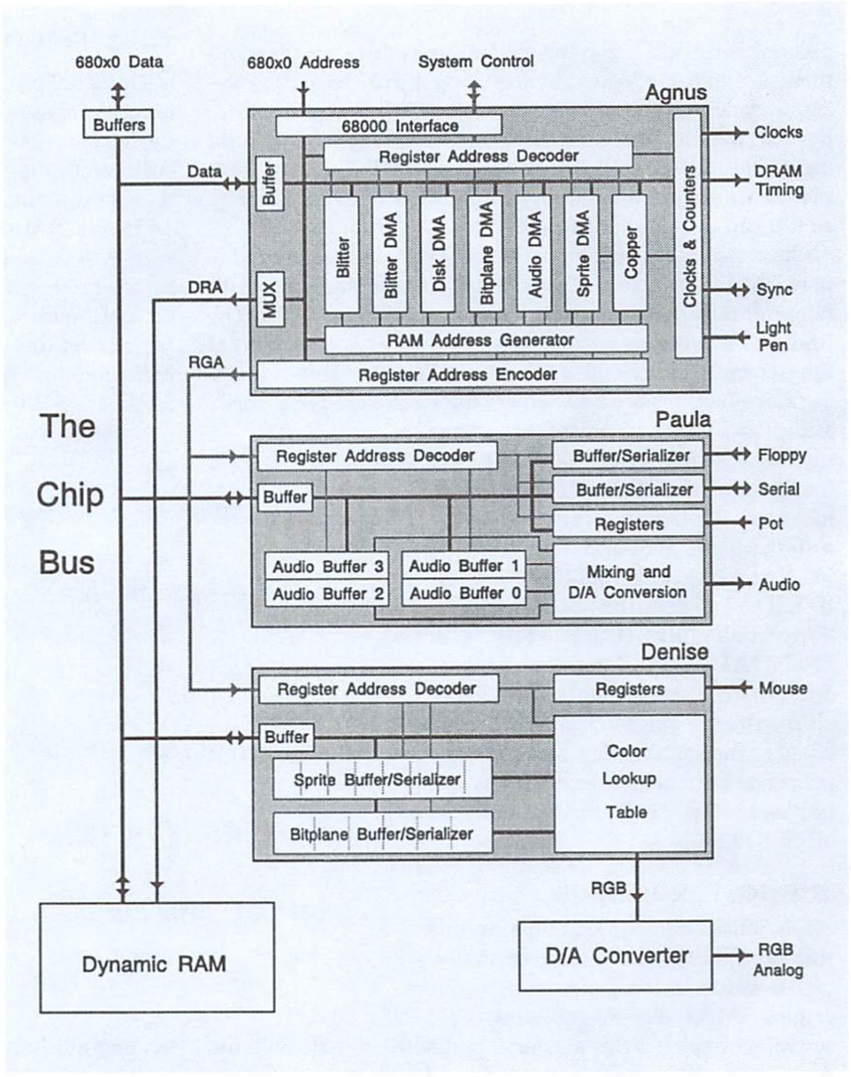


Figure 1. The chip-interconnect and internal chip functions of the Amiga custom chip set.

Chip DMA cycles are generated based on the register set-up of the various DMA controllers in Agnus. Consider a simple four-bitplane hi-res display. This requires Agnus's bitplane-control registers to be set properly for the chosen display, the bitplane DMA-fetch control to be enabled, and finally, four bitplane pointers to be loaded into Agnus's bitplane-fetch controller. Bitplane pointers are loaded via a Copper list, so Copper DMA must also be enabled. At the start of a display, before the visible portion, Agnus forces a "jump" to the primary Copper list pointer, which is given by a 32-bit Agnus-resident register pair, COP1LC ($080). The Copper fetches its first instruction, indicated by COPINS ($08C) on the RGA bus, which in this case is a move to BPL1PTH ($0E0), the high half of the first bitplane pointer. The next RGA cycle is the actual move, with $E0 on the RGA bus, the high half of the bitplane data value on the data bus. The Copper then sequences through fetches for BPL1PTL, then the high/low halves of BPL2PT, BPL3PT, and BPL4PT. Note that the Copper, like the 68000 (but not the Blitter), may access the chip ►

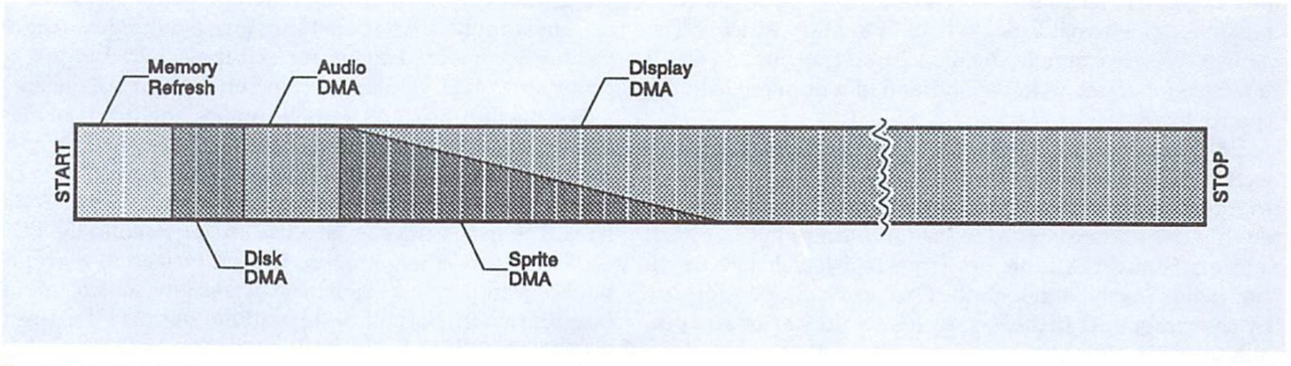


Figure 2. DMA on the Amiga chip bus.

bus only on "odd" chip-bus cycles, so in this case, four bitplane pointers are loaded by the Copper in 16 total roughly-280ns cycles, or about 4.48 microseconds. In a real-life Copper list, the bitplane configuration and color palette would most likely also be controlled by the Copper list. Once the bitplanes are set up, this example is finished with the Copper, so it'll end the Copper list with a wait for end-of-display.

Once the visible portion of the display is encountered, display DMA must take place. As usual, the scan line starts with memory refresh, optional disk, audio, and sprite DMA. The first and sometimes second refresh cycles have horizontal line strobe values ($038-$03e) on the RGA bus, the remaining refresh cycles are NOPs from the RGA viewpoint. Disk, audio, and sprite all have DMA channels on RGA when necessary, all based on DMA control and enable registers in Agnus. Bitplane data is fetched in the order plane 4, 2, 3, and then 1, so the RGA bus indicates BPL4DAT ($116), BPL2DAT ($112), BPL3DAT ($114), and, finally, BPL1DAT ($110). The BPLNDAT values don't access registers, but the display buffer in the Denise chip instead. The fetch of BPL1DAT triggers the serialization of all the appropriate buffers as determined by the bitplane setup. That's the basic idea of all chip bus DMA.

"Agnus was essentially just a chip-bus master.... Today's Agnus incorporates extra buffers and control functions on the same chip."

THE HOST PROCESSOR

Obviously, the Amiga chips do not run alone, but under the direction of a 680x0 family microprocessor. In the original A1000 design, Agnus was essentially just a chip-bus master; it had nothing to do with the particulars of the host-bus interface. External to Agnus, a number of buffers and PALs controlled 680x0 access to the chip bus under direction of Agnus's DBR* output, which indicates when odd cycles are being used by Agnus-managed resources.

Today's Agnus works very similarly, except it incorporates these extra buffers and control functions (the "fat") on the chip. In this way, a 68000 processor can just about directly hook up to Fat Agnus. The motherboard logic need only provide control for data bus buffer/latch circuitry, a pair of chip selects, and some kind of wait-state generator for the 68000 that monitors DBR*. This logic is generally incorporated in a gate array, such as the Gary chip used in the A500 and A2000. While the modern Fat Agnus defines the chip-bus-to-host-bus interface in terms of the 68000 microprocessor, it's quite possible, via external logic, to hook it to other systems. The Amiga 3000, for example, defines a 32-bit chip bus and 68030 interface for access with the addition of a bit more external control logic.

Fat Agnus translates 680x0 accesses based on two chip selects. One indicates that the access is to chip RAM and causes Agnus to generate a standard RAM-access cycle. The other chip select translates a processor address into an RGA address. Some RGA addresses access registers that the 680x0 can write directly (although the OS doesn't support this use by user programs). In theory, any RGA value can be sent out, although it makes little sense in most applications to drive DMA channels or other dynamic RGA-bus resources.

THE SERIAL PORT

The serial port is one of the simplest of the custom-chip interfaces. It has no DMA-control channel, and it is accessed via the host processor at all times. It's mainly controlled by three custom-chip registers located in Paula. The SERPER ($032) register contains a 14-bit number representing the number of 3.55 MHz (PAL) or 3.58 MHz (NTSC) clocks between bit transitions, plus one. While this lets a wide range of baud rates be set, interrupt lag times make excessive baud rates possible only with tight CPU read loops. The SETDATR ($018) register returns a single-level-buffered serial data byte, plus indicators for various error and buffer-state conditions. The SETDAT ($030) register transmits serial-data and format information as written to it. When SETDATR is full or SERDAT is empty, an interrupt can be generated through Paula's interrupt controller.

THE CONTROLLER PORTS

The mouse port, proportional-controller port, and light-pen port comprise most of the functions available at the standard "controller" port of an Amiga system. Each unit functions via simple programmed I/O. The two mouse ports are accessed via chip registers JOY0DAT ($00A) and JOY1DAT ($00C), contained in the Denise chip. The two bytes of each register represent vertical and horizontal counts of the mouse quadrature lines. The counters wrap in either incremental or decremental directions, so it's important that the CPU read each register often enough to prevent undetected wrapping. Usually once per vertical frame suffices for a 200DPI mouse under normal conditions. The mouse quadrature lines are defined to easily permit standard digital joysticks to be read. The actual hardware interface to Denise multiplexes both channels into one set of inputs to Denise, switched on the C1* clock (3.55 MHz or 3.58 MHz). This multiplexing rate is far more than sufficient to deal with mouse movement.

The proportional-controller interface is based on a Paula-resident set of four registers: POTGO ($034), POTGOR ($016), POT0DAT ($012), and POT1DAT($014). A "pot" device is a potentiometer, which, coupled with a precision capacitor on the Amiga motherboard, can form a time constant that is measured by Paula. A write to the POTGO register starts the measurement process. Both pot channels are grounded for approximately seven scan lines, then a current is applied. After each scan line, the charge on the capacitor is compared to an internal threshold, and the appropriate POTDAT counter is incremented if the charge is below the threshold. As with the mouse, the 16 bits of these counters are divided into vertical and horizontal eight-bit counts. Via configuration bits in POTGO, the pot lines can be set up as bidirection I/O lines instead of A/D conversion lines. The read values in this case are returned in the POTGOR register. When a mouse is used instead of a proportional controller, this mechanism is used for sensing mouse buttons two (right) and three (middle). A port bit in one of the CSG8520 chips is used to sense mouse button one (left).

The last of the simple port interfaces is the light-pen interface, which is based on Agnus registers. A light pen is designed to send a transition to the Agnus light-pen input when the screen pixel it is over is refreshed. That transition causes the state of the vertical and horizontal beam counters in Agnus to be latched in the VPOSR ($004) and VHPOSR ($006) registers. The vertical position, to an accuracy of one noninterlaced scan line and two low-resolution pixels, can be read from here. The counters are reset when the video display leaves the vertical-blanking interval, so it's important to read these during blanking for a stable result.

THE FLOPPY-DISK PORT

The disk interface (including control registers, DMA length, and DMA channel buffer/serializer) is located in the Paula chip, with the DMA pointer and counter in the Agnus chip. The controls in Paula allow for a variety of precompensation values in either GCR or MFM, with 4µs or 2µs per bit cell. On reads, DMA can be synchronized to begin on a match with a value stored in the DSKSYNC ($7E) register. Generally, all reads and writes are done one full track at a time; the disk hardware is not designed to deal with hardware sectors. The DMA capability supports large transfers without CPU intervention, and the full-track capability allows more logical sectors per track than with standard sector-oriented disk hardware. The general interrupt controller in Paula manages disk interrupts for "match with DSKSYNC" and for "DMA complete" conditions.

A full disk interface requires external control lines for such nondata-related disk functions as unit selection, head positioning, disk-removed sensing, and the various other elements of an industry-standard floppy-disk interface. These are handled by CSG8520 chips in Amiga systems. An index interrupt is also provided by way of a CSG8520 interface.

THE AUDIO SUBSYSTEM

Paula and Agnus house a considerable number of control registers that are associated with each of the four Amiga sound channels. Each sound channel has an Agnus-resident DMA-control unit to drive it from a sound-data table stored in memory. Sound data for any channel is a series of signed eight-bit values. The length of the sound table is also considered by the DMA controller, so the counter may automatically reset after playing through a sample, allowing a continuous sound to be played by each of the four channels with absolutely no processor intervention after setup. This allows the Amiga system to work as easily as computers with unattended "sound-chip" audio, as found in the C-64, without sacrificing the more flexible D/A sound-generation scheme that often requires constant CPU update on other systems.

Control registers allow a variety of options to be set for audio generation, and each basic parameter is available independently for each of the four channels. For volume control, a six-bit attenuation value (from 0 to –36.1dB) can be set. The frequency is expressed as a period based on the number of video clocks between samples. As shown in Figure 2, each channel gets one DMA cycle per scan line, which fetches two sample values. This sets a lower limit on the period of the sample of 34.642µs, which is equivalent to a maximum sampling rate of 28.867 KHz. This rate defines a Nyquest frequency of 14.334 KHz, about the frequency response of a decent cassette deck using CR02 tape (however, with only eight-bit samples, it won't have the dynamic range of cassette). Other registers permit channels to modulate each other in period and volume. As usual, the register address used by the audio DMA channels can be driven directly by the host CPU. The CPU can, of course, drive the audio-output channels at higher frequencies, because it's not limited to two samples per scan line. In addition, it can modulate the volume control for extended dynamic range, as the CPU to chip-bus bandwidth permits.

"The Amiga sound system works as easily as computers with unattended 'sound-chip' audio, as found in the C-64."

PLAYFIELDS AND THE COPPER

Display playfield management and the Copper are both located in Agnus. As discussed earlier, a main function of the Copper is to load chip registers, such as the bitplane pointers. As well as loading, the Copper unit receives input from the vertical- and horizontal-line counters (also in Agnus), which allow the Copper to wait for any particular position in a display. The interaction of WAIT and MOVE instructions is responsible for the sliding-screen feature you see under Intuition. Each change of screen involves a Copper wait, followed by a series of Copper moves to change display mode, color-table entries, and bitplane pointers. The space between screens is actually the result of the time it takes for all these Copper moves to run. The CPU is only involved when the user is moving the screens.

The main use of the Copper is in setting up bitplanes for display. Once these are set up, Agnus is responsible for generating bitplane-fetch cycles on the RGA bus, while Denise takes the fetched data, indexes it through the Color Lookup Table (CLUT), and ultimately generates 12 bits of digital-video output. The display hardware is extremely flexible. Registers in Agnus and Denise set up the size of a screen's pixels (139.68ns, 69.84ns, or 34.92ns), the horizontal and vertical size of the actual display, the bitplane line modulo for Agnus's display DMA counters, and such special modes as interlace, HAM, dual-playfields, and so on. When properly set up, Agnus generates the aforementioned DMA plane-fetch cycles (RGA address in the BPLNDAT range).

When Denise responds to a bitplane DMA cycle, it buffers up one word from each plane until there are enough to support the selected display mode. The pixel serializer routes pixels through special priority logic, which considers dual-playfields and sprites, and then to the CLUT, one pixel at a time. An interesting feature of the current Denise is that CLUT entries actually run at low-resolution pixel speed, 139.68ns. If the pixels are going at double speed ("high resolution"), the pixel stream input to the CLUT is actually multiplexed, so that every other pixel is from equally configured opposite halves of the CLUT. For ECS quad-speed pixels ►

("SuperHires"), pixel data is actually multiplexed after the CLUT. This accounts for the reduced number of SuperHires colors (64 rather than 4096) and the rather weird CLUT configuration the mode requires.

THE SPRITE HARDWARE

Sprites are small graphic objects controlled independently of the main display playfields. The system hardware supports eight sprites, each of which has its own Agnus-controlled DMA channel that fetches data in the scan line immediately before the main display fetch begins. Each sprite can be up to 16 139.68ns-pixels wide and two bitplanes deep. This is based on the sprite DMA channel, which fetches 32 bits per sprite per scan line. Because there is a sprite fetch on every scan line, sprites can be any number of pixels high.

Sprite bitplanes don't work exactly like display bitplanes. They're organized in memory so that only one sprite-data pointer is required for each sprite; the two WORDs for each scan line are consecutive in memory. Like bitplane pointers, sprite pointers are incremented as the sprite is serialized, so they need to be updated by the Copper at the start of every display. Also, only three of the possible combinations of sprite-pixel data access color registers; the fourth value always indicates transparency. Agnus also contains registers for sprite position and control. These are actually written by the sprite DMA channel before any sprite display occurs, greatly simplifying the software necessary for sprite management. The position and control registers set the horizontal start position, vertical start, and vertical stop position of each sprite. These values are compared with the value of the vertical and horizontal display counters, and the sprite engine starts sprite pixel generation when a comparator triggers it. CPU control of the sprite registers can allow the reuse of sprites on the display, in return for some 680x0 cycles spent.

"...the Amiga chips are still among the most sophisticated support chips used in any microcomputer."

The control register also permits a feature called attached sprites: The pixel data from two sprites is taken together to form a four-bit CLUT entry, allowing what appears to be one sprite supporting 15 colors and transparency. Attached sprites can move independently and only get taken together when they overlap. If they aren't attached, overlapped sprites normally overlay each other on the display, with sprite 0 in the front and sprite 7 at the back. Sprite overlay priorities are fixed, but a register in Denise supports rather flexible control of how the sprites interact with either of the two possible playfields, allowing sprite groups to sit in front, behind, or between the playfields.

THE BLITTER

Perhaps the most heralded feature in the Amiga chips is the Bit Image Manipulator or Blitter (called the "bimmer" by the original Amiga designers). This device, which occupies a substantial portion of the Agnus chip register and logic space, supports high-speed logical operations and moves on graphic images. The idea is that, with such hardware specifically designed for bit image manipulation, a basic 68000 system can stay low cost and still boast display performance characteristics of machines with several times more CPU power.

Agnus supports four DMA channels for Blitter data access, three for source operands (channels A–C), and one for destination (channel D). To support each channel, Agnus provides a separate pointer, pointer modulo register, and channel-enable bit. Additionally, channels A and B have barrel shifters, to perform bit-alignments without overhead. A common set of registers define the rectangular size of an operation, and the operation that will be performed. Any binary operation among the A, B, and C channels can be generated, with output to the D channel. Operations can include disabled channels that then act as constants with no DMA cycle. Additionally, Agnus pipelines the Blitter operation such that one Blitter read or write can take place every chip-bus cycle, as long as there are open slots on the chip bus to support this.

A few Agnus control registers allow modifications to the basic Blitter operation. To support blits of overlapping regions, the blit can be done with increment or decrement of the channel pointers and modulo. To handle end-point conditions on shifted blits, a pair of registers can supply end-point masks for the A channel. There are also a variety of area-fill modes in which the Blitter automatically fills the area on the selected sides of a series of single-pixel lines, with carry to support filling over multiple blits. There is also a special line-drawing mode that allows a patterned line to be drawn very quickly. The operation register is still active in this mode, so a number of different line operations are possible.

Considering that the maximum size of a blit is 1024×1024 pixels in the original Agnus (32K×32K in ECS Agnus), you should expect any given Blitter operation to take a relatively long time. To support parallel operation of the Blitter and host processor, the interrupt controller in Paula supports a relatively low priority "Blitter done" interrupt.

MORE TO EXPLORE

We've just scratched the surface of the Amiga chip subsystem. For more information on the register-level details of the Amiga chips, I recommend the *Amiga Hardware Reference Manual*, from Addison-Wesley. At over 350 pages, it still doesn't thoroughly cover every detail of the custom chip set.

In terms of resolution or number of colors, the Amiga chips may no longer be the hottest thing on the market, but they are still among the most sophisticated support chips used in any microcomputer. The basic architecture continues to hold up and shows promise for the future as well. ■

Dave Haynie was a senior engineer on the Amiga 2000, Amiga 2500, and Amiga 3000 computers and was the architect of the Zorro III expansion bus. He continues to be a driving force in Amiga system architecture and high-end design. Contact him c/o The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458, or on BIX (hazy).*

ON DISK

Easy File and Font Requesters

Try 2.0's ASL library (and our 1.3 alternatives) for ready-made interface help.

By Willy Langeveld

IN A MOVE towards standardization of interfaces, version 2.0 of the operating system provides a set of versatile file and font requesters via its ASL library. As you will see, you can easily tailor these requesters to fit the needs of most applications.

THE ASL LIBRARY

The ASL library contains three main functions. AllocAslRequest() and FreeAslRequest() respectively allocate and free an ASL file or font requester. Once a requester is allocated, AslRequest() displays it and allows the user to choose a file or font. The library's three additional functions—AllocFileRequest(), FreeFileRequest(), and RequestFile()—are provided only for backward compatibility with early versions of AmigaDOS 2.0x. As they were supplanted by particular invocations of the more general ...Asl... calls, Commodore discourages their use. The examples that follow use the three ...Asl... functions only.

The link library Amiga.lib (or the equivalent thereof for other C compilers) contains two other functions that interface to asl.library's offerings—AllocAslRequestTags() and AslRequestTags(). These differ from their nontags counterparts because they take a variable number of tag items as arguments, rather than a pointer to an array of tag items. Note, however, that you cannot use these functions if you link your program with cres.o under SAS/C, because they make absolute references to AslBase and IntuitionBase (as do other functions in Amiga.lib). In the Langeveld drawer of the accompanying disk, Example1 uses AllocAslRequest() and Example2 uses AllocAslRequestTags().

Before you plunge into coding, take note that asl.library resides in the Workbench 2.0x libs: directory, not in ROM. Therefore, the user might delete asl.library from his floppy to make more room; your application must be able to handle the possibility that it cannot open the library.

THE BASIC PROCEDURE

The basic procedure to bring up an ASL requester is to initialize an array of TagItems, allocate the type of requester you desire, display the requester, extract the needed information, and free the allocated resources. Consider the code required for a file requester (for a font requester simply substitute "font" for "file" when appropriate):

```
#include <libraries/asl.h>
...
struct FileRequester *freq;
/*
 * Substitute for N the number of tag items:
 */
struct TagItem tags[N+1];
BOOL result;
/*
 * Initialize tags array
 */
tags[ 0 ].ti_Tag  = ASL_...;
tags[ 0 ].ti_Data = (ULONG) ...;
...
tags[N-1].ti_Tag  = ASL_...;
tags[N-1].ti_Data = (ULONG) ...;
tags[ N ].ti_Tag  = TAG_END;
/*
 * Allocate the file requester
 */
freq = (struct FileRequester *)
   AllocAslRequest(ASL_FileRequest, tags);
/*
 * Put up the requester
 */
result = AslRequest(freq, NULL);
/*
 * Extract info from freq
 */
if (result) {
...
}
/*
 * Free the requester and all associated
 * resources.
 */
FreeAslRequest(freq);
```

As you can see, AllocAslRequest()'s arguments are a requester type and an array of tags. If you hand a NULL pointer to AllocAslRequest() instead of a tags array, you have a reasonable default file requester. The tag items merely override the defaults. AllocAslRequest() returns a pointer to a file requester or NULL, if it ran out of memory or failed for another reason.

Both AslRequest() and FreeAslRequest() take the pointer AllocAslRequest() returns as their first argument. AslRequest() also accepts another array of tag items as a second argument. This arrangement allows you to set up a file requester structure once using AllocAslRequest() and to modify only a few parameters in subsequent calls to AslRequest(). Also, all settings the user changes (such as the position and size of the requester) are retained if the same requester struc- ►

ture is used. I highly recommend this mode of operation. (Because the examples on disk use each requester only once, however, they do not take advantage of this feature.)

Note that AslRequest() returns a BOOL to indicate if the user selected a filename (TRUE) or cancelled the request (FALSE). Early documentation listed this function as returning a pointer to the filename the user selected, which is incorrect. The early versions of this function returned the filename, but without the associated path.

Clearly, I glossed over the most interesting parts of the process: how to initialize the tags array and how to get information back from the requester after the user has made a selection. Let's examine the tags issues first.

TAG ITEMS

Basically, a tag item is a pair of numbers—a tag and a value. The TagItem structure is:

```
struct TagItem {
   Tag     ti_Tag;
   ULONG   ti_Data;
};
```

where Tag is defined to a ULONG via a typedef statement. The ti_Tag field specifies the type of data represented by the tag item, and the ti_Data field contains or points to the data. (For more on tags, see "Tag Tips," p. 8, November/December '91.) The basic functions in the ASL library take an array of TagItem structures as a parameter. Each tag item describes a particular modification to the requester's default characteristics that are displayed. For example, to change the x location of the requester, which by default might come up in the top-left corner of the screen, you would include a tag item such as:

```
tags[4].ti_Tag  = ASL_LeftEdge;
tags[4].ti_Data = (ULONG) 100;
```

as part of the tags array in the code. The location in the array (in this case, 4) is unimportant.

All ASL parameters, structures, and tags are defined in the include file libraries/asl.h. The tags and a short description of the data they require follow.

Tags valid for both kinds of requesters:

Tag	Data
ASL_LeftEdge	Position of the requester's left edge.
ASL_TopEdge	Position of the top edge.
ASL_Hail	Pointer to a string of text for the title bar.
ASL_OKText	Pointer to text for the Okay gadget.
ASL_CancelText	Pointer to text for the Cancel gadget. Both the Okay and Cancel gadgets can display a maximum text size of six characters.
ASL_Window	Pointer to the parent window: The requester shares the IDCMP of this window unless instructed otherwise. Also, the screen to open on is determined by finding the screen the parent window is on.
ASL_FuncFlags	Bit mask with various flags, see later.
ASL_HookFun	Pointer to a call-back function. The ASL requesters can be instructed to call this function for each font or file to let the function determine whether the font or file in question should be displayed. The ASL requesters can also be instructed to call this function in case of IDCMP activity in the parent window.

Tags valid for the file requester:

Tag	Data
ASL_Width	Desired width of the requester.
ASL_Height	Desired height.
ASL_File	Pointer to a default filename. This filename is initially displayed in the requester's File gadget.
ASL_Dir	Pointer to the initial, default, pathname.
ASL_Pattern	Pointer to a default pattern string. All files (and, if requested, directories) are checked and dis played only if they match the pattern.
ASL_ExtFlags1	Bit mask with more flags, see later.

Tags specific to the font requester:

Tag	Data
ASL_FontName	Pointer to default font's name. This font is initially displayed in the requester.
ASL_FontHeight	Default font height.
ASL_FontStyles	Default font styles, see gfx.library's TextAttr structure for more details.
ASL_FontFlags	Default font flags, see the TextAttr structure.
ASL_FrontPen	Default pen number for the "front" color, the color in which the font's characters are drawn.
ASL_BackPen	Default pen number for the background color.
ASL_MinHeight	Minimum font height. Fonts shorter than this will not be displayed.
ASL_MaxHeight	Maximum font height. Fonts taller than this will not be displayed.
ASL_ModeList	Pointer to an array of strings to use for the drawing mode cycle gadget. This allows you to replace the entire cycle gadget including its label (Mode:). By default, the cycle gadget contains the mode JAM1, JAM2, and Complement. The first pointer should contain a pointer to a new label for the gadget. The last pointer should be set to NULL to indicate the end of the array.

Note: The width and height of the font requester cannot be specified. If the corresponding tag items are present, they are ignored.

FUNCTION FLAGS

You can also set flags using the ASL_FuncFlags tag item. For the file requester, your choices are:

Flag	Requester Action/Attribute If Flag Is Set
FILF_DOMSGFUNC	Calls the call-back function specified via ASL_HookFunc, with all the IDCMP messages that arrive at the parent window'sUserPort.
FILF_DOWILDFUNC	Calls the call-back function for each file to check if the file is to be included in the display.
FILF_MULTISELECT	Allows multiple files in the same directory to be selected.
FILF_NEWIDCMP	Uses its own IDCMP port; does not share parent window's.
FILF_PATGAD	Contains a pattern gadget.
FILF_SAVE	Changes the highlight state of the scrolling display, has the following properties: is not multiple-select, can create directories that are entered by the user and do not yet exist, does not support double-clicking on file names to prevent accidental overwriting of existing files. Use this flag for "Save" requesters.

For the font requester, the following flags apply:

Flag	Requester Action/Attribute If Set
FONF_FRONTCOLOR	Has a gadget to select the "front" color (color the text is drawn in).
FONF_BACKCOLOR	Has a gadget to select the background color.
FONF_STYLES	Has check boxes to select combinations of display styles (boldface, italics, underlined).
FONF_DRAWMODE	Has a cycle gadget for drawing modes (default JAM1, JAM2, and Complement).
FONF_FIXEDWIDTH	Displays only fixed-width fonts.
FONF_NEWIDCMP	Uses its own IDCMP port; does not share the parent window's.
FONF_DOMSGFUNC	Calls the call-back function specified in the ASL_HookFunc tag item, with all parent window IDCMP messages.
FONF_DOWILDFUNC	Calls the call-back function to inquire whether to include or exclude a font from display.

The file requester also has a couple of flags that you can set in the ASL_ExtFlags1 tag item:

Flag	Requester Action/Attribute If Set
FIL1F_NOFILES	Is a directory requester. No files are displayed, no File gadget included.
FIL1F_MATCHDIRS	Checks directory names during pattern matching.

I'll discuss individual flags in more detail as we encounter them in the examples.

THE FILE- AND FONTREQUESTER STRUCTURES

The structure returned by AllocAslRequest() is either a struct FileRequester or a struct FontRequester depending on the type asked for. The public fields in the file requester structure are:

```
struct FileRequester {
   ...
   BYTE   *rf_File;
   BYTE   *rf_Dir;
   ...
   WORD   rf_LeftEdge, rf_TopEdge;
   WORD   rf_Width, rf_Height;
   ...
   LONG   rf_NumArgs;
   struct WBArg *rf_ArgList;
   APTR   rf_UserData;
   ...
   BYTE   *rf_Pat;
   ...
};
```

With one exception, these fields should *not* be written to by the application—they are of interest only after the AslRequest() function has returned. For a single-select file requester, the filename is returned in the rf_File field and the path (if any) in the rf_Dir field. The path returned is precisely what was last displayed in the Drawer gadget in the requester, so it may be a full path specification, a relative path, or nothing at all if the file is in the "current" directory. Similarly, the filename is what last appeared in the File gadget. Note that this means that any restrictions that were in effect during the display of the requester (whether in the form of a pre-supplied wildcard pattern or imposed by the call-back function) need to be checked again by the application: The user may have typed a nonconforming file- or pathname anyway.

The next four fields contain the position and size of the requester as it was just before the user closed it. As a matter of user-interface style, it may be desirable to remember these values and use the corresponding tag items to initialize the next invocation of the same requester to the same numbers. If you use the same requester structure in subsequent calls to AslRequest(), this is automatic.

For a multiple-select file requester, the files are returned as a WBArg list—the same kind of argument list that is present in the WBStartup message sent to applications when they are started from the Workbench. The number of files in this list is stored in the rf_NumArgs field and the names of the files and locks on their home directories are returned in the array of WBArg structures to which the rf_ArgList field points. Note that only the filenames themselves are in this list. You can either concatenate the names with the contents of the rf_Dir field to get complete (relative) pathnames, or you can use the lock in the WBArg structure to find the file and use NameFromLock() to obtain the full pathname. The latter method is preferred (more on this in the Example2 discussion).

You can write to the rf_UserData field—an exception to the rule. Typically this is useful for nonblocking requesters where the application needs to keep track of the context from which the requester was called. Example2 uses this field to pass the parent window pointer to the call-back function in a reentrant way. This field needs to be initialized *before* the call to AslRequest().

Finally, the rf_Pat field contains the last pattern that was present in the Pattern gadget, if such a gadget was displayed. You can instruct the file requester to list only files matching a certain pattern in two ways: by specifying the pattern either as part of the ASL_Dir tag item or separately with an ASL_Pattern tag item. The pattern in the ASL_Dir tag item will not be displayed anywhere, even if there is a Pattern gadget. Yet, only files matching the pattern will be shown. If an ASL_Pattern tag item is specified, the requester will show only the files that match that pattern, and if there is a Pattern gadget, that pattern will be shown in it. If both pattern-matching methods are used together, then all files displayed match both patterns. Note that you can use the ASL_Pattern tag item without a Pattern gadget, but, the user cannot change the pattern in this case.

For the font requester structure, the public fields are:

```
struct FontRequester {
   ...
   struct TextAttr fo_Attr;
   UBYTE  fo_FrontPen;
   UBYTE  fo_BackPen;
   UBYTE  fo_DrawMode;
   APTR   fo_UserData;
};
```

The fo_Attr field is a complete TextAttr structure that contains the values for the selected font. You can clone it and use it directly in a call to OpenFont() or OpenDiskFont(). Specifically, the name of the selected font is in the ta_Name field of this TextAttr structure. The other fields in the FontRequester structure contain the front pen, back pen, and drawing mode selected by the user, respectively. The front and back pens are returned as pen numbers and the drawing mode is returned as 0 for JAM1, 1 for JAM2, and 2 for COMPLEMENT mode (see graphics/rastport.h), unless you substitute your own ►

list. If you do, the sequence number of the item in the list is returned, with 0 the first item in the list.

Again, these fields are *read-only* by the application, with the exception of the fo_UserData field, which you can write to as in the case of the file requester.

ASL IN ACTION: EXAMPLES ON DISK

The Langeveld drawer of the companion disk contains:

Example1	The first example program.
Example2	The second example program.
Example1.c	The main routine and initialization for the first example.
Example2.c	The main routine and initialization for the second example.
makefile	The dependencies file for the two programs.
MyARP.h	An abbreviated ARP include file.
requesters1.c	The ASL and ARP requester interfaces for Example1.
requesters2.c	The ASL requester interfaces for Example2.
simpreq.c	A simple string requester.
util.c	A variety of utility functions used in the examples.

All C files are compiled using SAS C with 16-bit ints (-w option). This is to ensure that the examples will run that way, not because it is necessary or even desirable. It does point out, however, the need to cast tag item values to ULONG in various places, so this is a useful exercise.

"Under 2.0, the program uses the ASL requesters, ...under 1.3 it brings up the file requester available in the ARP library."

Example1 provides basic examples of typical file, directory, and font requesters, including all the code you need for a "complete" user interface and to handle the problems you might encounter under AmigaDOS 1.3 and 2.0. Under 2.0, the program uses the ASL requesters, while under 1.3 it brings up the file requester available in the ARP library. If neither is present, Example1 falls back to a simple string requester. It is written in a modular fashion, so you can compile requesters1.c, util.c, and simpreq.c and link the object modules to your application with a minimum of work. The function calls are very simple and take care of all the details, including frequently omitted style issues such as blocking user-input in the parent window and displaying a busy pointer if the parent window is activated. In addition to handling the different OS versions transparently, Example1 makes sure that DOS requesters and the requesters themselves come up on a visible part of the proper screen.

When you run Example1 from the Shell, you see a usage note and the first requester. The usage note explains that the program takes up to four command-line arguments: window, screen, usearp, and noarp. When no arguments are specified, the program cycles through the three requesters and indicates whether something was selected and if so, what.

If the window argument is specified, a "parent" window will open before the first requester. This window can be resized and has both a close button and a regular Intuition gadget. When the first requester appears and you click in the parent window, the mouse pointer becomes the standard Amiga busy pointer. When you try clicking on the gadget in the parent window, nothing happens--all Intuition input to that window is blocked. Clicking on the Close gadget will not result in a message sent to that window, although resizing the window prompts a NEWSIZE message when the requester closes. The requesters in this example are "blocking" requesters that comply with the *Amiga User Interface Style Guide*'s (Addison-Wesley) recommendations for "modal" requesters.

If the screen argument is also present, a new screen is opened first. The window and its associated requesters then open on this screen and everything works as before. Note, that if the screen argument is present but the window argument is not, the requesters open on the Workbench and the screen covers everything. It is *not* possible to open ASL requesters on screens if they do not have a parent window, with the exception of the current default public screen. With the ARP requester this is possible, because the ARP requester can be set to call a call-back function with the requester's NewWindow structure. (I use this feature of ARP in Example1, but only to set the requester position, not to make it open on the screen).

If the usearp argument is present, the example "simulates" the absence of asl.library. In this case, the program displays a system requester allowing the user to cancel, retry, or go on anyway. This provides an example of using the new EasyRequest() function (which is not part of asl.library). Of course, this will only happen under AmigaDOS 2.0x. A retry causes the program to continue as if the user copied asl.library back to his libs: directory. When "Go On Anyway" is selected, the program uses a combination of the ARP file requester and the simple string requester.

Finally, when noarp is specified in addition to usearp, the program simulates the absence of both libraries and only simple string requesters are used.

DISSECT THE SOURCE

Looking at Example1.c, you'll find a number of includes and then the prototypes for the three main functions in requesters1.c: GetFileName(), GetDirName(), and GetFontName(). The main program begins by calling the function Intro(), which parses the command line arguments and returns a set of flags. Intro() also prints out the usage message. The flags are used to call the function OpenStuff().

OpenStuff() deserves a few remarks. The part of interest is:

```
... AslBase = OpenLibrary("asl.library", 0L);
if (AslBase == NULL) {
   ...
   if (IntuitionBase->LibNode.lib_Version >= 36) {
      while (AslBase == NULL) {
         reply = EasyRequest(NULL, &CheckForAsl, NULL, NULL);

         if (reply == 0) goto cleanup;
         if (reply == 2) break;

         AslBase = OpenLibrary("asl.library", 0L);
      }
   }
   if (AslBase == NULL) {
      ...
   }
}
```

If the ASL library cannot be opened, OpenStuff() checks if it's running in 2.0 (Intuition's version is 36 or larger). If it is, it opens a system requester via EasyRequest() to ask the user what to do. The EasyRequest() structure, CheckForAsl, is initialized at the top of the function. The requester has three buttons; if the user clicks on Cancel (reply is 0) the program quits, if the user clicks on Go On Anyway (reply is 2), it breaks out of the loop without opening ASL, and if the user clicks on Retry, Example1 tries opening the ASL library again.

Next OpenStuff() opens a screen that has the 2.0 3-D look:

```
if (flags & SCREEN) {
    ...
    if (AslBase) {
            testscreen = OpenScreenTags(NULL,
            SA_Depth,    (ULONG)  3,
            SA_Pens,     (ULONG)  penarray,
            SA_Title,    (ULONG)  "ReqTest",
            SA_DisplayID, (ULONG) HIRES | LACE,
            TAG_END);
    }
    else {
    testscreen = OpenScreen(&ns);
    }
    ...
}
```

The trick here is to use OpenScreenTags() with the SA_Pens tag item included. The penarray was declared earlier in the file to be the minimal form acceptable for this purpose:

```
static UWORD penarray[] = {0xFFFF};
```

Under AmigaDOS 1.3, OpenStuff() opens a screen using OpenScreen() and a NewScreen structure.

Other functions in this file are CloseStuff() and CheckIDCMP(). The former closes all the opened resources, and the latter checks whether any messages arrived on the parent window's IDCMP.

Back to the main function: It contains three "exercises." The first calls GetFileName(), whose arguments are pointers to the parent window, a filename buffer, a pattern, and a "hail" string. The filename buffer may be initialized to a default value, in this case foo.foo. If the user cancelled the requester, GetFileName() returns FALSE, and a descriptive message is printed. Note that in this case the default filename is not overwritten. If the user selected a file, that filename is displayed. PostMsg() prints to the parent window if one was opened, otherwise it prints to the Shell from which the example was launched. After the exercise, the program checks the parent window's IDCMP to see if messages arrived. Similarly, the second exercise brings up a directory requester by calling GetDirName(), and the third displays a font requester by calling GetFontName(). Let's take a closer look at how these functions work.

REQUESTERS1.C

The first thing you notice in requesters1.c is that GetFileName() is just a wrapper for calls to routines specific to the ASL, ARP, or simple string requesters. Next are calls to the functions Set/ClrWindowBusy() and SetTaskWindow(), which are described in more detail later. SetWindowBusy() sets a busy pointer for the parent window and blocks out Intuition activity. SetTaskWindow() causes DOS requesters to appear on the proper screen while the file requester is up or, alternatively, causes them to not appear, if it is called with an argument of –1. At the end of GetFileName(), both changes are undone. The ASL requesters themselves do not take care of these details for you.

The function GetFileNameASL() starts by copying the file and path parts of the default filename into buffers called fil and dir. These buffers have a fixed size of 256 bytes—AmigaDOS is still limited to pathnames that are 255 characters or fewer. (While AmigaDOS can support directory nestings whose full path specification is larger than 255 characters, no single path specification can be longer.) The splitting is done using the new AmigaDOS functions FilePart() and PathPart(), which each take a complete filename as an argument. They behave almost identically, the only difference being that FilePart() returns a pointer to the beginning of the filename part of the string, whereas PathPart() returns a pointer to the : or / just before the filename, if either exists.

AddPart() adds the pattern specification, given in the third argument of GetFileNameASL(), to the path part. The requester the program brings up does not have a pattern gadget, but lists only files matching the pattern. Note, that the pattern is not displayed anywhere in the file requester.

This example tries to open the requester at an x, y location of 150, 50. If the user has previously dragged the screen almost all the way down, however, the requester normally would be invisible. If there is a parent window (as in most applications), then the functions find_bestx() and find_besty() try to compensate for the position of the screen and modify the suggested values they take as their second argument. Note that this is of great importance with large overscanned screens or the 2.0 AutoScroll screens. Ideally speaking, the requesters should appear more or less inside the outer borders of their parent window. The find_bestx/y() functions don't currently do this, but could be enhanced with a bit of work.

Finally, the tags array is initialized and the requester is allocated and displayed using the basic procedure described earlier. Notice that the flag FILF_NEWIDCMP is specified, making this a blocking requester. After the requester returns, the path- and filenames are extracted and combined into a single filename, if the user actually made a selection (result being TRUE in that case).

GetFileNameARP() cannot use the various ...Part() functions, because it is presumably running under AmigaDOS 1.3. The ARP library does contain functions that do similar things, but in these examples I wanted to concentrate on ASL and to minimize the number of ARP functions used. Roughly equivalent code separates the file and path parts and adds them together again.

The function GetDirName() is very similar to a merged version of GetFileName() and GetFileNameASL(). The important difference is that the ASL_File tag item is missing and that an ASL_ExtFlags1 item is added: It sets the FIL1F_NOFILES flag, making this a directory requester. It looks considerably simpler, mainly because no temporary buffers are needed and no paths and files need to be assembled or disassembled. There are currently a few bugs in this form of the requester. The Path gadget does not automatically get activated, and when a name is typed in the gadget and the user presses return, the requester does not close: The user must click the Okay gadget or use the equivalent menu item. These problems are likely to be fixed in the future.

Finally, we have the function GetFontName(). For demonstration purposes, this font requester lists only fixed-width ►

fonts that come in at least two sizes: eight pixels wide by eight pixels tall and eight pixels by 11 pixels. The tag items ASL_MinHeight and ASL_MaxHeight, plus the FONF_FIXEDWIDTH flag in the ASL_FuncFlags tag item specify part of this, the rest has to be done with a call-back function. This function, FontHook(), is specified in the ASL_HookFunc tag item and announced in the ASL_FuncFlags tag item using the FONF_DOWILDFUNC flag. After the requester returns, the program finds the font name in the TextAttr structure contained in the FontRequester structure.

FontHook() is called by the font requester with three arguments: the type of the call (the reason the function was called), a pointer to an object, and a pointer to the file requester structure itself. In case the type is FONF_DOWILDFUNC, the object is a TextAttr structure. Because the TextAttr structure doesn't contain sufficient information for the requirements of the example, it calls CheckFont(), which opens both the desired fonts based on the font name and checks that they satisfy the conditions. CheckFont() is in util.c.

UTIL.C

The function SetWindowBusy() sets the 2.0 or 1.3 AmigaDOS busy pointer for the parent window of the requester. It also puts up a "real" (Intuition) requester in that window. That requester has, however, no size (0 pixels wide and tall), so it is completely invisible. It does have the effect of completely blocking all IDCMP traffic to that window. If an application must have a blocking requester that is really a window (as is the case with the ASL requesters), the *Amiga User Interface Style Guide* recommends the actions performed in this function. ClrWindowBusy() undoes the actions of SetWindowBusy(). Respectable sources have recommended using a 1-pixel-wide and -tall requester located relative to the window at position (–1, –1), but this overwrites random memory in some cases and should, therefore, *not* be used.

Next, CheckFont() calls GetFont(). This function attempts to open a font when given a TextAttr structure and a width and returns a pointer to the font. CheckFont() itself tries to obtain 11×8 and 8×8 versions of a font and returns failure if either or both don't exist.

The functions find_bestx() and find_besty() determine the location of the left and top edges of the screen (ViewPort, actually) the window is currently on. If the screen is not in the normal position, the functions try to compensate by returning adjusted values for the requester position.

SetTaskWindow() solves a problem with DOS requesters. The Process structure of each process on the Amiga contains a default-window pointer that is normally set to NULL. With this pointer AmigaDOS determines where to place its own requesters (such as "Insert volume foo: in any drive"). Most often, these appear on the Workbench, even if the application has a screen of its own. When an application does something that might prompt such a requester, it should set the process-window pointer to the main window on its screen. The DOS requesters will then appear on this screen. Notice also that the process window must at all times be a valid window pointer or else when a DOS requester needs to open, the system will crash. For this reason, always restore the process-window pointer to the original value as soon as possible. SetTaskWindow() sets the process-window pointer to the one given as its argument and returns the previous value of the process-window pointer. A second call to SetTaskWindow() with this old window pointer restores the original value. When the process-window pointer is set to –1, DOS requesters are not displayed. For demonstration purposes, I set the process-window pointer to –1 if there is no parent window and to the parent window if there is one.

"The Process structure of each process on the Amiga contains a default-window pointer that is normally set to NULL."

The next functions, AddList() and NukeList(), are used only in Example2. AddList() adds a node to an already existing Exec list with the node name set to a copy of the string passed as a second argument. If the list doesn't exist yet, a new list is created. The function returns a pointer to the list or NULL if it ran out of memory. NukeList() takes a list allocated by AddList() and deallocates it.

The last two functions in util.c are a case-insensitive compare called strcmpu and a rindex function that behaves slightly differently from the ANSI strrchr function.

SIMPREQ.C

Simpreq.c contains the code for a simple string requester with Okay and Cancel gadgets. It was generated quickly using PowerWindows and wasn't designed to be particularly pretty. If you prefer another string requester, you can probably substitute it for this one with minimal work. About the only noteworthy piece of code in simpreq.c is the section that decides whether to open on a custom screen or on the Workbench:

```
if (window) {
   if (scr = window->WScreen) {
      if (((scr->Flags & SCREENTYPE)
            == CUSTOMSCREEN) ||
            ((scr->Flags & SCREENTYPE)
            == PUBLICSCREEN) ) {
         nw.Screen = scr;
         nw.Type   = CUSTOMSCREEN;
      }
   }
}
```

Note that you must check not only if the screen the parent window is on is a CUSTOMSCREEN, but also whether it is a PUBLICSCREEN. In both cases, however, the NewWindow's Type field should be set to CUSTOMSCREEN.

EXAMPLE2: ADVANCED ASL

Demonstrating the ASL requesters' more advanced features, Example2 runs under 2.0 only, takes only window and screen as command-line arguments, and exits if it cannot find the ASL library. It displays a regular file requester, a multiple-select file requester, and an elaborate version of the font requester. All three are "nonblocking," the type of requester preferred by the *Amiga User Interface Style Guide.* As with Example1, if the window argument is specified, a "parent" window opens, then the first requester appears. Now, however, when you click in the parent window, the mouse pointer does not change and clicking on the window's close button, Intuition gadget, or resizing the window, causes messages to be displayed in the window. This is accomplished using ASL's call-back facilities.

Example2.c is almost the same as Example1.c. The differences are that OpenStuff() requires asl.library to be present and that the three requesters are called GetFileName1(), GetFileNames(), and GetFontName(). The calling sequence for GetFileName1() is the same as for GetFileName() in Example1.

GetFileNames() is a multiple-select file requester and it returns a pointer to the struct List. Complete filenames are present in the ln_Name fields of the nodes in this list. The code illustrates how to loop over the nodes and print out the selected filenames. Once the filenames are printed, the list is deallocated using NukeList() (see util.c).

The calling sequence for GetFontName() is identical to that of the function with the same name in Example1.

Instead of using requesters1.c, Example2 employs requesters2.c. Its basic differences from requesters1.c are that it uses AllocAslRequestTags() instead of AllocAslRequest() and if there is a parent window, the requesters are nonblocking and share the parent window's IDCMP. In fact, this is a requirement: To be nonblocking, the requester *must* share the IDCMP port of the parent window. In this case, the program calls SetTaskWindow(), but not SetWindowBusy().

For the first example, GetFileName1(), the changes are that the FILF_NEWIDCMP flag is not set, but the FILF_DOMSGFUNC flag is. In addition, the ASL_HookFunc tag item is set to a pointer to FileHook(), which is elsewhere in this file. Just for fun, I added a Pattern gadget and made this a "save" requester. Note that the case in which the parent window does not exist (the pointer is NULL) is automatically handled by the requester—it allocates its own IDCMP and everything continues to work. Also note that before the call to AslRequest(), the rf_UserData field is set to the pointer to the parent window.

If there is a parent window and, for example, the user clicks on the gadget while the requester is up, FileHook() is called, much as FontHook() was in Example1. Here, the object is a pointer to a struct IntuiMessage and the application can do whatever it needs to based on the class of the message. The example program simply writes a text message about what happened to the parent window. It can do this because it retrieves the pointer to the parent window from the requester's rf_UserData field.

The requester also has the FILF_DOWILDFUNC flag set, so for each file FileHook() is called with a pointer to an AnchorPath structure. All information about the file should be extracted from this structure, not from the file-requester structure. Specifically, the filename can be extracted from the ap_Info field, which is a complete FileInfoBlock structure. A pointer to the lock on the directory that contains the file is available in the an_Lock field of the AChain structure pointed to by the ap_Current (or, equivalently, ap_Last) field of the AnchorPath structure. FileHook() will accept all the files; the code is here only as an example of how to be more selective than is possible using pattern matching about the files that are listed. If you need to change directories inside this function, be sure to get back to the initial directory before returning.

GetFileNames() is the multiple-select file requester. It was remarkably simple to make it multiple select: I added the FILF_MULTISELECT flag to the ASL_FuncFlags tag item. Retrieving the selected files is, however, a bit more complicated. The "easy" way is:

```
nfiles = freq->rf_NumArgs;
for (i = 0; i < nfiles; i++) {
   strcpy(dir, freq->rf_Dir);
   AddPart(dir, freq->rf_ArgList[i].wa_Name, MAX_PATHLENGTH);
   filelist = AddList(filelist, dir);
}
```

The number of files is returned in the rf_NumArgs field. The example uses this to loop over the files and assemble the path and file parts into a full filename. The file portion is stored in a list of WBArg structures. AddList() then generates an Exec list of full filenames that can be returned to the caller. Notice that the full filenames are still in principle "relative" paths (relative to the current directory), just as with the other file requesters.

The limitation to the multiple-select requester is that it only allows selection of multiple files from the same directory. This is implicitly assumed in the way the file and path parts are assembled in the above code. If the restriction is ever removed, the method breaks. I therefore recommend you use a slightly more complicated procedure to obtain guaranteed absolute paths: Change to the directory of the file, obtain a lock on the file and use the function NameFromLock(). The relevant code is:

```
nfiles = freq->rf_NumArgs;
for (i = 0; i < nfiles; i++) {
   oldlock = CurrentDir(freq-> rf_ArgList[i].wa_Lock);
   if (oldlock) {
      lock = Lock(freq->rf_ArgList[i].wa_Name, ACCESS_READ);

      if (lock) {
         NameFromLock(lock, fil, MAX_PATHLENGTH);
         UnLock(lock);
      }
      CurrentDir(oldlock);
   }
   filelist = AddList(filelist, fil);
}
```

The last example in requesters2.c is a more elaborate version of the Example1 font requester. A few flags and an ASL_ModeList tag item were added. The example, however, doesn't do anything with the selected font styles, modes, or colors. The main difference is really in FontHook(), which now deals with the parent window IDCMP messages. Notice that for IDCMP messages, both FileHook() and FontHook() must return the pointer to the object, because the requester still needs to reply the message to Intuition.

FINAL REMARKS

The basic operation of the ASL requesters is remarkably simple. On the other hand, the requesters are very flexible and can be equipped with lots of bells and whistles. After reading this article, you should not have any excuses left for not using ASL requesters in your applications. ■

Willy Langeveld is a physicist and scientific programmer at the Stanford Linear Accelerator Center (SLAC). He is the author of a.o. VLT and rexxarplib.library, plus is the moderator of the Amiga.user conference on BIX. Contact him c/o The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458 or on BIX (langeveld).*

Libraries, Structures, and System Calls

By Jim Butterfield

THE AMIGA IS crammed with advanced features that many programmers can't reach. All the knowledge of BASIC, C, Modula-2, or even assembly language can't provide that elusive link to the Amiga's inner workings. Another element is needed: learning how to tap the Amiga's architecture.

Your program must communicate with the Amiga by means of calls to libraries, devices, and resources. With such calls, you may create, investigate, or modify a host of activities within the Amiga. Elements such as screens, windows, gadgets, menus, messages, files, processes, and more all will go to work at your command.

No matter what programming language you choose, you need to learn how to access the Amiga's rich roster of features. Programming textbooks often focus on the language itself, with limited attention to system architecture. In this column I will try to fill that gap, emphasizing the Amiga's architecture, and deemphasizing the particular programming language you may be using. Examples—complete programs or fragments—will be given in two languages, principally BASIC and C.

C: GENERIC VERSUS CUSTOM

Programs written in C compile into efficient machine language code. While many C programmers write at a relatively low level (close to the machine's architecture), C comes with functions that permit faster program writing and more portable code. Each programmer chooses a compromise between the two approaches: code that's "close to the machine" versus code that uses standardized C functions.

I call the two extremes custom and generic coding. Most programmers use some of each. Generic programming produces programs that are easily transportable, but that take little advantage of the Amiga's special resources. Custom programming exploits the Amiga's dazzling operating system, but creates programs that often will not travel easily to other systems.

Generic-style programs tend to use C's standard functions. Memory is allocated with malloc(), and messages are printed with printf() or putchar(). Files are handled in "level 2" style, using fopen(), fclose(), and I/O statements such as fread(), fgetc(), fputc(), or fwrite().

Custom-style programs tend to address Amiga library functions directly. Such programs might allocate memory with the Exec library function AllocMem(); with it, you may specify the type of memory desired, such as chip or fast. Messages would be printed using the DOS library function Write(). Files will be opened and closed with Open() and Close(), again from the Dos library.

Because my intent is to show how to access the Amiga's inner workings, I plan to emphasize custom-style Amiga C programming.

BASIC: IN TRANSITION

BASIC is designed principally as an interpreted language. That usually makes it easier to code and debug, but slower to run. Many developers find BASIC a good language to rough out logic flow and test concepts. When the logic looks sound, the program may be rewritten into a language such as C. The best known Amiga Basic is...Amiga Basic, of course. For many years, Amiga Basic came with your Amiga at no extra charge. With new Amigas, you must buy it separately. Other implementations of BASIC are available for the Amiga: F-BASIC (Delphi Noetic Systems), True BASIC (True BASIC Inc.), and COMAL (Comal User's Group), to name only three. Each has its enthusiasts, and all of them are capable of being used to explore the Amiga's inner workings.

INVOKING THE SYSTEM

Your programs will typically interact with the Amiga system by means of two activities. They will call subroutines located in the Amiga's shared libraries; and, as part of these calls, they will often make use of structures. Normally, you won't peek and poke memory addresses on the Amiga. Instead, you politely ask the system, via its library function calls, to do various jobs for you.

"Shared library" is a phrase that many beginners find confusing. The word library calls to mind a place where you look things up. On the Amiga, shared libraries are much more than that: Each is a collection of function calls. The programmer often views such a library as a "jump table;" each item in the table is a subroutine-call address that invokes a specific Amiga operation. A close cousin to the library is the device. Again, the word is misleading; in this context, device does not mean the hardware. Instead, a device is a set of programs to handle input and output for a specific piece of hardware.

The various libraries have names that seem familiar to Amiga users. You may recognize the following names: Exec, the master control point of the Amiga; Dos, a major handler for input and output, particularly where disk drives are involved; Intuition, a central clearing house for screens, windows, keyboard, and other user interfaces; and graphics, a facility to help draw into screens or windows. There are many other libraries, some of which we'll use only occasionally. In total, there are thousands of system calls that we can make to the various libraries.

When we call one of the many system library functions, we may need to deal with structures. A structure is a collection of information. To exploit the Amiga, you must use data stored within the various structures.

Consider some examples of structure usage. If we call for information about a disk file, using the Dos library function Examine(), the result will be supplied to us as a structure called a FileInfoBlock. This structure contains such items as the filename, its protection bits, size, comment field, and other elements. A program needs to know how to look at this structure; for example, that the first character of the filename is located eight bytes from the start of the FileInfoBlock structure.

Here's another example: Suppose you wish to create a new window. To do so, your program calls the Intuition library function OpenWindow(). Before making this call, however, you must set out details of the desired window, including: its size, its position, and the gadgets it have. All this, plus other information, must be built into a structure called NewWindow; that structure, in turn, is supplied to OpenWindow(), which creates the window as specified.

Once you've learned to call a function within a system library and how to deal with structures, you're well on your way to tapping the Amiga's power. You can go about these activities in several ways, depending on the programming language. In the final analysis, however, they are the same system calls and the same structures no matter what language is used.

"Once you've learned to call a function within a system library and how to deal with structures, you're well on your way to tapping the Amiga's power."

THE C APPROACH

To use the functions of any library, you must first connect to it with the function OpenLibrary(). When you're finished with the library, release it with CloseLibrary(). The C language automatically opens two of these for you.

The pre-opened libraries are Exec and Dos; their reference points (library bases) are stored as variables SysBase and DosBase. Many programs won't need to use these variables directly; the C system will call upon themwhen appropriate.

All libraries other then Exec and Dos must be opened with an call to OpenLibrary() and, eventually, closed with CloseLibrary(). The pointer (library base) returned by OpenLibrary() must be stored under its proper symbolic name; C will use that name when it's asked to call one of that library's functions. The symbolic names are easy to remember; the following code fragment gives the idea.

```
struct IntuitionBase *IntuitionBase;
struct GfxBase *GfxBase;
...
IntuitionBase = (struct IntuitionBase *)
OpenLibrary("intuition.library",0L);
GfxBase = (struct GfxBase *) OpenLibrary("graphics.library",0L);
... (main program) ...
CloseLibrary(GfxBase);
                    CloseLibrary(IntuitionBase);
```

Here's something that confuses beginning C programmers: IntuitionBase, GfxBase, and similar items have two separate identities. First, the name is used to describe (in C terminology, declare) a structure; the description usually comes from an include file such as intuition.h. Secondly, the same name defines a pointer that the program needs to store within its memory.

I'll try to clarify this double meaning by describing the code above. The first line of code might be read as: "Set aside memory for a pointer, which will point to a structure of type IntuitionBase; call the pointer IntuitionBase." C understands the two kinds of usage without confusion. You might find consolation in the thought that this kind of thing simplifies the job of choosing a name for the library base pointers: The pointer name is the same as the structure name.

The same double-purpose name usage is found in the OpenLibrary line. Reading the line roughly in reverse, we might translate as follows: "Open a library called intuition. library, any version. When you get the library-base pointer, note that it points at a structure of type IntuitionBase; finally, store the pointer into variable IntuitionBase."

About structures: C usually gets declarations for Amiga structures from the appropriate include file. When you need to handle a structure, check your documentation for names of the various fields. ►

Remember that you must get names of libraries and functions exactly right, including upper or lower case as necessary. You won't find a library called Dos.library or a DOS function called delay; in both of these, the letter D is in the wrong case.

ABOUT AMIGA BASIC

Amiga Basic ("Microsoft Basic for the Amiga") was developed by Microsoft Corporation. The language is fully featured and fairly fast for an interpreted language. Amiga Basic also has a number of features tailor-made for accessing the inner workings of the Amiga. Libraries are accessed with the command LIBRARY, and return values from specific function calls are given "types" by means of DECLARE FUNCTION... LIBRARY. Structures are most often built with two keywords: SADD (String Address), and VARPTR (variable-pointer). Keep in mind that BASIC strings are not normally terminated with CHR$(0); frequently you must tack on this character before using SADD to set up a pointer to a string. Structures are often built as BASIC integer arrays; a pointer to their location within memory produced by VARPTR.

"...you must get names of libraries and functions exactly right... You won't find a library called Dos.library or a DOS function called delay; in both of these, the letter D is in the wrong case."

LIBRARY signals that the program wants to use one of Amiga's shared libraries and causes Amiga Basic to issue OpenLibrary() to the system.

For LIBRARY to work, a bmap file must have been created for the named library. A bmap file, such as dos.bmap or graphics.bmap, should be located in the current directory or, preferably, in libs:. To create a bmap file, look on your Extras disk, in the drawer BasicDemos. You'll find several bmap files, plus the program ConvertFD, which makes bmap files. Create the ones you want and copy them to your libs: directory.

By the way, ConvertFD gets its information from a set of files named "...fd" that are located on the Extras disk. These are text files: Read them! They provide quick documentation on the function calls of each library. For example, dos_lib.fd holds information on the system calls that we are about to make here: Open(), Close(), and Execute(). For more detailed documentation on dos.library calls, consult *The AmigaDOS Manual, Third Edition* (Bantam Books).

FIRST PROJECT

This first programming technique is a quick trick. It's a way of triggering a CLI/Shell command directly from your program. Execute(), in the DOS library, performs any command as if it were typed into a CLI window. The function must be provided with an output file handle (that's an AmigaDOS term). We get that by opening a file to device NIL:, which gives us a handle to "nowhere."

The usual method for a program to acquire a disk directory involves many calls to functions within dos.library; it can get a little tedious. You must first call Lock(), then Examine(), and then make repeated calls to ExNext(); each call gets one directory item. Finally, a call to UnLock() releases the directory. With version 2.0 of the operating system, the work can be done using the new function ExAll();, but either way, it's a grunt job. We're going to skip all that by using Execute(); a CLI command will write a complete directory listing to a file of our choosing.

Incidentally, 2.0 has an interesting alternative to the Execute() function. The new function SystemTagList() does the same job and allows many options. This example, however, employs Execute(), which works on all Amigas.

THE BASIC SOLUTION

A couple of quick notes, and then we'll go straight to the code. Keywords OPEN and CLOSE already exist in Basic; so dos.library function calls Open() and Close() are renamed as xOpen() and xClose(). Functions Open() and Execute() both return a value to the calling subroutine; we must assign a "type" to this value using a DECLARE FUNCTION statement. Close() doesn't return anything of interest to us, so it doesn't need DECLARE FUNCTION.

Now, take a look at the code:

```
REM - This demo program shows how to
REM - invoke Amiga's DOS library
REM - EXECUTE routine from Amiga Basic

' We must use Open/Close to allow
' operation from Workbench start!
DECLARE FUNCTION xOpen& LIBRARY
DECLARE FUNCTION Execute& LIBRARY
' No need to declare xClose
' Here comes the code
LIBRARY "dos.library"
   z$ = CHR$(0)
' Open NIL: as a file to allow output path
' Invoke the dos.library Open (as Basic xOpen) function
' handle = Open(filename,mode)
   handle& = xOpen&(SADD("NIL:"+z$),1005)  ' MODE_OLDFILE
   IF handle&<>0 THEN
' Invoke the dos.library Execute function
'success = Execute(commandString, input, output handle)
      x = Execute(SADD("list >RAM:temp"+z$), 0, handle&)
' the file handle to NIL: has done its job; close it
' Invoke the dos.library Close (as Basic xClose) function
' Close(handle)
      xClose(handle&)
' List the directory in RAM:temp
      OPEN "RAM:temp" FOR INPUT AS 1
      WHILE NOT EOF(1)
         LINE INPUT #1,a$
         PRINT a$
      WEND
      CLOSE
      KILL "RAM:temp"
```

Continued on p. 55.

Exclusive ffer From migaWorld

TOOL CHEST

ONLY $12.95 EACH
BUY 2 – GET 1
FREE

Quality Software at an Affordable Price!

Special "Tool Chest Theme" Disks.

#TSP1. Maps of USA, Canada and Europe Hi-Res IFF brushes of all 50 States, 10 Canadian Provinces and countries in Europe. Each map has a 3D extrusion effect with cast shadow.
#TSP2. Sounds Animal sounds, musical instruments and miscellaneous sounds. 43 sound samples in all.
#TSP3. Games1 *3D TicTacToe.* A challenging 3 dimensional tictactoe game. *Crystal Caverns.* A colorful, graphic adventure game. *Dot2Dot.* An adaption of the classical 2-player connect-the-dot game. *PDMan.* A fast maze game where you deliver pizzas before the monster gets them. *TinyBall.* The world's smallest baseball arcade game. *Surround-Cycles.* A high-speed arcade game, loosely based on the light cycle races from the movie Tron.
#TSP4. Games2 *Dyno-Wars.* A strategic checker-style game. *"Kooties".* Assemble a bug before the computer. *Blockbuster.* Get rid of blocks by pushing matching blocks into them. *"Warrior".* A fun shoot'em-up adventure game. *Trilon.* A tetris-style game. *Circe.* Battle the computer's armies to take over the planet Circe.
#TSP5. Games3 *CrossCircuit.* A one- or two-player action logic game. *Word Search.* Generate your own word search puzzles. *Drugbusters.* Get rid of cigarettes, alcohol, pills and needles. *Cinco.* Try to get five X's in a row before your opponent does. *Color Logic.* A Master Mind-like game.
#TSP6. 3D Vector Objects #1 Lamp, mirror, bed, chair, fire hydrant, lamp post, street lamp, street sign, traffic light, cannon, cannon supplies, rose, fork, spoon, knife, plate. All objects are in Sculpt format.
#TSP7. 3D Vector Objects #2 Books, Early American telephone, barrels, gun, spaceships, lamps, trombone, bugle, skateboard, Lego-Rover, remote control, glider airplane. All objects are in Sculpt format.
#TSP8. Clipart B&W and colorful clipart for a variety of needs. People, holiday clipart, school items, scrolls, food, monsters, fish, lobster, symbols, and more.
#TSP9. Utilities #1 *IconMeister.* Icon editor with a complete set of tools. *TJFormat..* A disk formatting program. *Chartmaster.* A graphing tool that allows you to generate and save different graphs. *Graph3D.* Visualize complex data in 3-D manner and save the results. *StrucGen.* Create gadget, text, bitmap, and window structures and write the C source code to disk. *LabelMaker.* A very easy-to-use label program. *DoublePrint.* Prints your text files on both sides of the paper.

#TC11. (Sept/Oct 89). *DiskSalvage* will fix corrupted disks and recover deleted files. *IconMeister* is an advanced Icon editor. *AlienDuel* is a fast, shoot'em-up game. *Crystal Caverns* is a colorful, graphic adventure game. Plus an animation, sound samples, 3D dinner table objects, and Hi-Res monster clipart.

#TC12. (Nov/Dec 89). *TJFormat* formats disks that AmigaDOS chokes on. *ChartMaster* is a powerful graphing tool. *MoreCandy* generates an assortment of colorful graphic patterns and saves them to disk. Shark, a game where you have to eat all the little perch you can catch to stay alive. Plus B&W clipart, spaceship 3-D vector objects, an animation, and Speakeasy (C source code for the programmer who wants to add speech to C programs).

#TC13. Jan/Feb 90). *StrucGen* lets you create gadget, text, bitmap, and window structures and writes the C source code to disk. *Lifecycles* plots your biorhythms. Create spectacular images with *Mandelbrots. Look*, a CLI utility that helps you find files on your disk quickly and easily. Plus 3-D objects, holiday clipart, an Amiga Flight animation, drive head cleaning utility, and a technical discussion on how AmigaDOS stores information on disk.

#TC14. (March/Apr 90). *Amigo File* is a database & addressbook. *Graph 3-D* lets you visualize complex data in 3-D manner and save the results. *3-D TicTacToe. Spectrogram* analyzes and displays digitized sound. *Pteronadonaball*, an animation by Lou Markoya. Plus 3-D objects, food clipart, and sound samples.

#TC15. (May/June 90). *Labelmaker* is a very easy to use label program. *SurroundCycles*, a high-speed arcade game. *Dot2Dot*, an adaption of the classical connect-the-dot game. Animation by Brad Schenck that also is a tutorial on animation techniques. Collection of clipart, plus animal sounds.
#TC16 (July/Aug 90). *Cyberscape* generates complex 3-D vector objects of terrain and saves your objects in Sculpt or Turbo Silver format. *Mathplotter* graphs mathematical equations in both 2-D and 3-D. *Double Print* prints your text files on both sides of the paper. Plus *Metamorph* (a graphic adventure game), *TinyBall* (the World's Smallest Baseball Arcade game), and six digitized sound effects.

#TC21. (Sept/Oct 90). *Circe* is a "Risk-style" game where you battle the computer to take over the planet Circe. *Batchman* allows you to execute CLI programs, batch files and ARexx scripts by clicking on a gadget. *Colors* prints color test sheets and assigns RGB values to printer and screen output. *Easyfile* is a powerful database for home or small business. *Sprite Editor* lets you create animated sprites. Plus 3-D fruit objects, and five sound samples.

#TC22. (Nov/Dec 90). *Amiga Checkbook* is a fast checkbook manager with check printing ability, budget analysis, and a report writer. *AmiCrypt* is a file encryption/decryption tool. *WaveSyn* lets you design instruments and waveforms for use in other programs. It also lets you play your keyboard like a piano. *Cinco* is a game where you try to get five Xs in a row before your opponent does. *AlterCLI* lets you change attributes of your CLI window with a single command. Plus an animation by Lou Markoya, and 3-D objects.

#TC23. (Jan/Feb 91). *WhereIs* lets you quickly find a misplaced file on your drive. *Select* creates a window from which you can run your favorite programs. *MatheMagic* lets you find the day of the week for any date in the century, lets you enter statistical data to be analyzed, graphs data, and solves linear algebraic equations. *Pointer List Maker* lets you create your own pointers or import pointers from other programs and save them in a disk file. *Dyno-Wars* requires more strategy and thought than your regular checker board game. Plus *Mr. Monster* and *Trilon* games, 3-D objects, six sound effects, and a Pretzel animation.

#TC24. (March/Apr 91). *TableMaker* is a tool for building tables of all sorts. *"Kooties"* mis a fun game where you have to assemble a bug before the computer beats you to it. *HT (HyperText)* lets you put links in a text file to other text, graphics, sound, animation, music, ARexx, and other applications. *The Little Black Book* is a telephone and address book. *Color Logic* is a Master Mind like game. The computer picks the colored pegs that you have to guess the positions of. Plus four disk and memory utilities, and animal sounds.

#TC25. (May/June 91). *ScreenTolff* captures your screen to an IFF file. *PopColors* lets you change the colors of your Workbench screen. *"Warrior"* is a fun shoot'em-up adventure game for one or two players. *Kaleidoscope* is a line drawing program. *Blockbusters* is a game where you get rid of blocks by pushing matching blocks into them. *Amidemo* demonstrates how fast the Amiga does solid-polygon animations. *Microscope* lets you examine memory at any location in the Amiga. Plus Hi-Res IFF brushes of all 50 states and 10 Canadian provinces, nine digitized sounds, and two small disk utilities and one printer utility.

#TC26. (July/Aug 91). *Word Search Maker* lets you create word search puzzles. *TimeCalc* is useful for anyone who has to calculate hours and minutes. *Drugbusters* is a fun game where you get rid of cigarettes, booze, pills and heroin needles. *Wordfinder* lets you search binary or text files for a particular word or string. *KeyClick* will add a small click to your keys when pressed and *MouseClick* will add a small beep to your mouse. *Seawar* is a game between the Allied Navy and the Empire Fleet. Plus Early American 3-D vector objects, digitized sound effects, and a simple DIR program.

#TC31. (Sept/Oct 91). *Calendar Publisher* allows you to personalize and print calendars. *CrossCircuit* is a one- or two-player action-logic game that calls for fast reflexes and fast thinking. *Apr* is a flexible printer utility. *Pizza Delivery Man* is a fast maze game where you deliver pizzas in your neighborhood. *MultiPlayer* displays IFF pictures or brushes as well as IFF sound samples. Plus *RAMGauge*, 3-D vector objects, and Hi-Res IFF brushes of all maps and flags in Europe.

#TC32. (Nov/Dec 91). *Computer Coloring Book* comes with 15 pages of pictures to color. *Listmaker* lets you make, edit, combine, and print lists of words. *Moresmooth!* displays text files on the screen. *Pocket Billards* lets you choose from several of the most popular pool games. *Texture* lets yoou create, manipulate, and save realistic fractal textured surfaces. Plus *Plague* (monster maze game), banjo and guitar 3-D objects, *Screen Fader, SnowBench*, and *FlipBench.*

#TC33. (Jan/Feb 92). *Measures1.0* converts measurements from unit or system to another. *DiskLabeler* is a database designed for storing and printing 3.5" disk labels. *Jewels*, a game of action and strategy. *Switch* a fun board game. *GFXclipper* captures and saves graphic screens. *Softball Manager*, a database for storing stats for players, teams and leagues. *WB Exec* allows you to execute CLI commands from WorkBench. Plus *Clipart* and an index of all the back issues of AmigaWorld Tool Chest.

ORDER FORM

Write your selection below and enclose this form with proper payment. Only $12.95 each.Buy two get one free!

#____	#____	#____	Total disks ____ x $12.95 each **(Buy 2 get 1 free)**	$ ________
#____	#____	#____	Add $3.50 postage/handling for each order (Canada & Mexico $5.00, Foreign orders $11.50)	$ ________
#____	#____	#____	Canadian orders add 7% GST	$ ________
#____	#____	#____	California Residents add 7.25% Sales Tax	$ ________
			Total enclosed	$ ________

☐ Check enclosed
Charge my: ☐ MasterCard ☐ Visa
☐ Amex ☐ Discover

CARD# ________ EXP. DATE ________

SIGNATURE ________

NAME ________

ADDRESS ________

CITY ________ STATE ________ ZIP ________

Tool Chest • P.O. Box 802 • 80 Elm Street • Peterborough, NH 03458
1-800-343-0728 or 1-603-924-0100
2CBB2

ON DISK

Screens for Public Consumption

By John Toebes

PUBLIC SCREENS ARE a very important but often overlooked feature of OS 2.0. If you use VLT or any other program that relies on screenshare.library, you're probably already familiar with the concept of a program opening up its window on the screen of a program other than Workbench. The 2.0 version of Intuition makes this a standard part of the programming paradigm.

From the simplest view, public screens have three basic users:

• Applications that create their own screens and hence are targets for other windows to open on them. In some cases, you might even *expect* particular applications to be opened on the screen.
• Applications that open windows that might be placed on other than the default Workbench screen.
• A public screen manager that allows the user to control these interactions. This category, of course, is very limited in the number of possible applications (how many screen blankers do you expect to run?), while the other two cover just about every program that can be written.

To the user public screens can be a real benefit. Applications that previously could not be displayed at the same time (such as an editor and a terminal package) can now be combined easily at the user's discretion. This, combined with the existing ARexx support, allows for some truly integrated applications.

Fortunately, public screens were designed with the normal programmer in mind. There is very little that an application has to do to support them, but you must give them careful thought.

CREATING A PUBLIC SCREEN

By definition, public screens mean that any program can open up on a screen. From the screen side there are few implications, but the application creating the screen and windows must be aware that other windows may overlap its windows. For this reason, you can no longer assume that things such as front-to-back gadgets are unnecessary. The most common mistakes are:

• Using a SIMPLE_REFRESH window with NOCAREREFRESH set, because you assume that no other window will be present.
• Using the SCREEN bitmap for rendering directly.
• Not including front-to-back or sizing gadgets when other windows occupy the same screen.
• Creating a BACKDROP window and expecting it to be the topmost or only window.

Fortunately, these are easy to check for and remedy. The first one requires changing to either a SMART_REFRESH window or implementing appropriate refresh routines. Applications that use the SCREEN bitmap are probably not suitable candidates for public screens. The last two have some window presentation implications, but the 2.0 look and feel brings enough changes that you probably have to change your application in this area anyway.

In addition to ironing out these issues, your program must also include a couple of new routines to create a public screen. The most important is OpenScreenTagList(). Tags are the 2.0 way of passing many of the new optional parameters to system routines. OpenScreenTagList() accepts three tags:

• **SA_PubName:** Indicates the name of the public screen. If this tag is not present, the screen is not public. If it is present, the ti_data field should point to the NULL-terminated name for the screen.
• **SA_PubSig, SA_PubTask** (must be give as a pair): The task and the signal bit to use to notify you when the last window on the public screen is closed. Make sure that they appear in the tag list after the SA_PubName tag.

Although not specific to public screens, the SA_ErrorCode tag is useful for figuring out errors. The tag's ti_Data field is a pointer to a LONG that holds the return code from OpenScreenTagList(). If you do not supply this, you will have no idea why the screen failed to open.

As you can see, a public screen is identified by its name. Because of this, each public screen must have a unique name. Generating it is probably the hardest thing to do for an application—particularly when you may wish to run multiple copies of the application at one time. How you solve this problem is dependent upon how you expect the application to run. Obviously, a hardcoded name is not a good choice. Your alternatives are:

• If the public screen is already open, ignore the failure to open another public screen of the same name and default to letting the application use the previously opened screen instead of the current one. For such programs as text editors and word processors, this may be a very good implementation.
• Permute the name by adding a count to the end or the current time, and then try again.
• Fail the operation. If you really can allow only one copy to

run at a time, this will certainly guarantee it.

• Make the name of the public screen the same as the ARexx port of your program. This method is extremely appealing because most programs now have to implement ARexx ports and already solve the name-uniqueness problem for them. Most likely your ARexx port name was generated by permutations or even by a user command-line option, so you can avoid duplicating code. This has the added bonus of providing a common external name that other applications can use in interacting with your application.

With the above requirements fulfilled, all you need to open the screen is a routine such as:

```
static ULONG oserror;
static char name[50] = "AW_PublicScreen";

/* I accept the default pens, but have to
 * pass something to get the new look.
 */
static UWORD sa_pens[] = {
   0, 1, ~0     /* just detail and block */
};

struct TagItem nsext[] = {
   { SA_PubName,  (ULONG) ps_name },
   { SA_ErrorCode, (ULONG) &oserror },
   { SA_Colors,   (ULONG) colorspecs },
   { SA_FullPalette, (ULONG) TRUE   },
   { SA_Pens,    (ULONG) sa_pens },
   { TAG_DONE,          }
};

static struct ExtNewScreen ExtNewScreeen =
{
   0, 0, STDSCREENWIDTH, STDSCREENHEIGHT, 2, /* top, left, width,
                                                height, depth */
   0, 1, HIRES, CUSTOMSCREEN,   /* Detail, block, viewmodes, type */
   NULL, NULL,            /* font, title (use Pubscreen name) */
   NULL, NULL, NULL          /* gadgets, bitmap, extension   */
};

struct Screen *openPubScreen()
{
   struct ExtNewScreen ns;
   struct Screen   *screen;
   int count = 0;


   while ( screen != OpenScreenTagList( &ns, nsext ) )
   {
      if (oserror != OSERR_PUBNOTUNIQUE)
         return(NULL);
      /* Someone else has the name, so try for the next one */
      sprintf(name, "AW_PublicScreen_%d\n", count++);
   }
   PubScreenStatus( screen, 0L );  /* make the screen available */
   return ( screen );
}
```

This code is very similar to how you would open an existing screen, but adds logic to generate a unique name and a call to make the screen publicly available. Public screens have both a public and a private status; typically you make a public screen private when you are attempting to close the screen down. If you want to do some preparation on the screen before allowing other windows to open, you may wish to delay the call to PubScreenStatus(). The key to remember is that once you switch a screen to public status, you must assume that anyone (and possibly everyone) can open up on the screen.

CLOSING A PUBLIC SCREEN

Opening a public screen is the easy part. When you have several visitor windows on your screen and you want to terminate the application the trouble really begins. Your choices for a solution are:

• Attempt to close the screen as normal. If an error occurs (because there are visitor windows), display a requester and retry the operation after the user selects OK. This is the approach that the 2.0 Workbench uses.

• Use the SA_PubSig and SA_PubTask tags on OpenScreenTagList() to make your task wait for the signal before shutting down. In this way it can do a friendly wait before cleaning up. You still have the general problem of keeping a potentially large program in memory just to close a screen, however.

• Spawn a separate process to wait for the signal and terminate the window when the signal is received. This allows your main application to terminate without worrying about the window, but requires a bit more work in getting the separate process running. One nice thing about this process is that it can be shared by multiple applications.

```
static struct EasyStruct myezreq = {
   sizeof (struct EasyStruct), 0,
"Close Public Screen",
   "Please Close All Windows on this Screen",
```

►

```
    "OK"
};

void closePubScreen(screen)
struct Screen *screen;
{
   PubScreenStatus( screen, 1L ); /* make the screen unavailable */
   while (!CloseScreen(screen))
   {
      /* The screen didn't close, prompt the user for a chance to try
      again */
      EasyRequestArgs( NULL, &myezreq, NULL, NULL );
   }
}
```

OPENING A VISITOR WINDOW

Applications that want to open windows on a public screen have very little set-up work to do. They need only specify the public screen name they wish to open on when they make the call to OpenWindowTagList(). Like OpenScreenTagList(), OpenWindowTagList() recognizes a few tags:

- **WA_PubScreenName:** Specifies a public screen to open on by name. The ti_Data field points to a NULL-terminated string.
- **WA_PubScreen:** Specifies a public screen to open on by address. In this case, ti_Data points to the actual Screen structure. You must make sure that the screen will not close during the call.
- **WA_PubScreenFallBack:** A Boolean that indicates whether Intuition should use the default public screen (or Workbench) if the named public screen isn't available.

"Applications that want to open windows on a public screen have very little set-up work to do."

When you pass a screen by the address, you must ensure that the screen is not going to close. To accomplish this you can lock the screen open with the LockPubScreen() call or be certain that a window that won't go away is already opened on it.

To lock a public screen, call LockPubScreen(name), passing the null-terminated name of the public screen as name. If the screen exists, you will get a lock on it and the function will return the pointer to the screen. Otherwise you will get a NULL pointer. Once you have safely opened your window on the screen, you must call UnlockPubScreen (name, screen) to release the screen so that it can be eventually freed.

Given the ease of opening a window on a public screen, you should also add a little piece of power to your programs to take full advantage of public screens—screen jumping. Be warned: Accomplishing this task might take a bit of code restructuring and introduce some issues you may not be ready to tackle. The most important of which—resolution and font independent code—is beyond the scope of this article. As your application window jumps from screen to screen, you can see complete differences in layouts because of larger fonts or screen resolutions. Jumping to a lower resolution screen may cause you to have to seriously shrink a window.

To prepare for jumping, your application needs to be able to close down the window at any time and then reopen it. If you put all your initialization and termination into a single place, this is easy. If you have it all mixed in with the general initialization code, it might take a while to get it all staight. You must also handle how the user specifies the jump operation. Do you jump to a specific screen by name, to the "next" available screen, or to a screen selected by the user (such as the front-most screen). The choice of implementing this is up to you, but remember that the *user* is requesting the operation, so it should be built in as a fundamental part of the user interface.

Once you have determined that the user wants you to jump to a new screen (probably by a key sequence or menu operation), you need to close down the window. Next, locate the screen to jump to with NextPubScreen() and lock it. Now, reopen the window on that public screen and unlock the screen. Because it is possible for the screen to disappear between the time you identify it with NextPubScreen() and the time you actually lock it, you can completely eliminate the lock and unlock steps by taking advantage of how OpenWindowTagList() works:

```
void dojump()
{
   char buf[150];

   /* Make sure we can find a new screen before closing the window */
   if (NextPubScreen( curpubscreen, buf))
   {
      /* As everything is now modular,  just close the old window and */
                        /* open the new one on */
                        /* the new screen. */
                        CloseMyWindow();
                        OpenMyWindow(buf);
            }
         }
         struct TagItem wintags[] = {
            { WA_PubScreenName,   (ULONG) NULL },
            { WA_PubScreenFallBack, (ULONG) TRUE },
            { TAG_DONE,            }
         };

         void OpenMyWindow(name)
         char *name;
         {
            struct Window *win;
            wintags[0].ti_Data = (ULONG)name;
            ... Other window structure information based
            on application ...
   win = OpenWindowTagList(&nw, wintags);
   if (win == NULL)
   {
      /* ooops, we can't get the window open, we might want to go
      back to */
      /* a previous screen (if we can) or just fail the application.  */
      /* This is a sticky situation that needs to fit the application.  */
      /* remember that you might not even be able to reopen the window on */
      /* the original screen.              */

}
```

Remember that your application needs to handle the changes in the screen behind the window. If you have cached colors, they will certainly change, as well as aspect ratios and

Continued on p. 55

Official Home of the Commodore Technical Team

If you're after technical know-how, there's only one place to get the answers you need *fast* and that's on **BIX** (BYTE Information Exchange).

BIX is the official home of the Commodore Technical Team of prominent Amiga developers and Commodore Applications and Technical Support staffers.

Tap into the Amiga Technical Team on BIX in these conferences:

amiga.user	Exchange ideas, solve problems, compare notes
amiga.sw	Amiga programming and developer issues
amiga.hw	Amiga hardware design, use, and hookup
amiga.arts	Artistry using the Amiga
amiga.int	Developing for the international Amiga
amiga.special	Special guests and events
amiga.unix	Unix on the Amiga
amiga.games	Games on the Amiga
amiga.com	Commodore's conference for commercial developers
amiga.dev	Commodore's conference for all developers
amiga.world	Amiga World magazine
aw.techjournal	Amiga World Technical Journal

Check the Fishdisk library index and make an inquiry for the programs you want. Search extensive databases of Amiga "how to" reference materials and find remedies to your problem. Or join in on "how to" sessions designed to show you how to get more out of your Amiga.

Call BIX *today*, the official home of the Commodore Technical team and by far the most technically oriented on-line service for Amiga users.

BIX McGraw Hill

One Phoenix Mill Lane Peterborough, NH 03458 800-227-2983 (voice)

ON DISK

Spritely Rendering

By Leo L. Schwab

OCCASIONALLY, YOU MAY need to put dynamically rendered imagery into a sprite. That is, a static image in chip RAM won't do; you need to actually draw it. Unfortunately, because of the way sprites are represented in memory, using the rendering functions in graphics.library is, at best, nonobvious. Once you understand how to attach a RastPort to a sprite, however, you can easily perform standard rendering operations.

THE PROBLEM

Figure 1 illustrates the primary difficulty. The Amiga graphics functions are designed to operate on bitplanes that are scattered throughout memory, and each plane must be a complete unit. Sprites, on the other hand, are a different animal. The bitplanes for a sprite are sort of stuck together sideways. A single line of sprite imagery is arranged as pairs of WORDs. The first WORD in each pair represents the low-order bits of the sprite pixels (plane zero, so to speak), the second represents the high-order bits (plane one). The sprite hardware puts these together and displays three colors plus transparency. What's more, each sprite has a little "cookie" at the top and bottom, which contains positioning control information for the sprite hardware. You definitely don't want to draw into those areas.

Somehow, you must "split" the pairs of image WORDs so that graphics functions can operate on them as if they were separate planes. You also need to keep from drawing over the control WORDs.

MAPPING THE BITS

The solution to attaching a RastPort structure starts with a BitMap structure, so we'll begin there.

The BitMap must be formatted such that each line in each bitplane begins in exactly the same horizontal position as the previous line. In other words, for the first WORD in an image pair, we want to describe a BytesPerRow width that, after advancing that many bytes in memory, will leave us at the first WORD of the next image pair.

At first glance, this might seem to be two bytes (one WORD), because that's the width of a single plane. Advancing two bytes, however, would leave us in the second WORD of the current image pair, which isn't right. If we were to advance four bytes (two WORDs), we would skip over the second WORD in the image pair and end up at the first WORD in the next pair, which is precisely what we want.

For the second WORD in an image pair, the increment is exactly the same; two bytes to take us to the next image pair, and two more bytes to skip to the second WORD in that pair. Thus, we should initialize our BitMap structure to have a BytesPerRow value of four, or 32 pixels.

The height of the BitMap is set to the height of the visible sprite imagery. The position-control WORDs are not included in the height. The depth of the BitMap is two planes.

Finally, the plane pointers must be initialized. Plane 0 is set to point to the first WORD of the top line's image pair (past the position-control WORDs). Plane 1 points to the second WORD of the top line's image pair.

It sounds complicated, but it really isn't. In C, the whole thing is done as follows:

```
struct SimpleSprite *spr;
struct BitMap  *spritebm;
PLANEPTR  spritedata;

InitBitMap (spritebm, 2, 32, spr->height);
spritedata = (PLANEPTR) spr->posctldata;
spritebm->Plane[0] = spritedata + 4; // Past position control
spritebm->Plane[1] = spritedata + 6; // Second word of image pair
```

HOW IT WORKS

Creation of the BitMap is the core of this trick, so it's worth spending a little extra time on this point to understand why this works.

If you've read the *Amiga Hardware Reference Manual* (Addison-Wesley), or Tom Rokicki's articles on the Blitter (such as "The Complete Guide for the Blittering Idiot," p. 2, October '91), you know that the Blitter operates on rectangular regions of imagery. To do this, the horizontal and vertical dimensions of the area are described to the Blitter so that it knows how much work to do. This is why the BitMap structure exists. It describes the width and height of the imagery to graphics.library. Graphics then takes these values, converts them into the appropriate hardware values, and feeds them to the Blitter.

Often, however, you don't want to operate on the whole image, just a small part of it. The Blitter can be told to touch only the part you want and skip over the rest. The amount to skip is called a modulo. When doing blits, graphics.library figures out how much to skip by subtracting the width you specify from the total width of the bitmaps affected.

The fields BytesPerRow and Height in a BitMap structure describe the total size of a bitmap. Graphics.library makes particular use of BytesPerRow to calculate modulo values. For example, if you did a blit operation two bytes wide into a bitmap six bytes wide, the graphics.library would program the Blitter (in the simplest case) to write two bytes of data and skip four

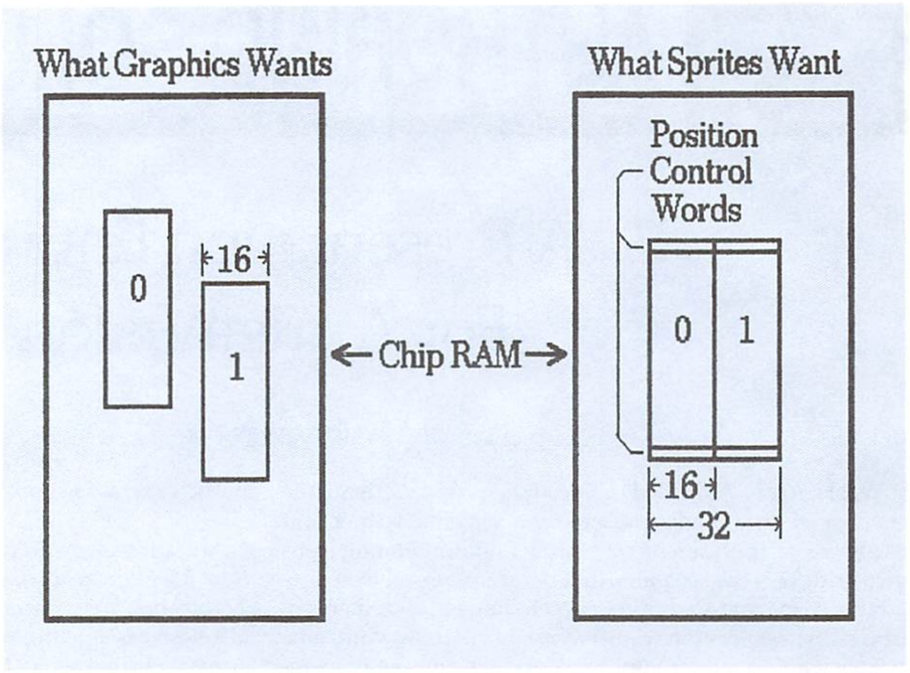


Figure 1: Differences in memory representation.

bytes to get to the next line of imagery. This is the core of the technique—getting graphics.library to set the modulos up just right.

Recall that sprite images are two bytes wide, but that it takes four bytes to advance from one line of imagery to the next. So, we tell graphics.library that our bitmaps are four bytes wide. The fact that only the first two bytes of each line are valid doesn't matter; we just make sure we don't draw outside the valid area. In addition, it doesn't matter that the bitplanes we described overlap each other in memory. Neither graphics.library nor the Blitter care if the planes overlap. They treat each plane as a single unit no matter where it is. By restricting our rendering to the first 16 pixels, graphics.library will draw into the first two bytes of each plane and skip the other two.

The final effect is that the individual WORDs of the image pairs are written to separately, as if they were ordinary planes. Thus, normal rendering is accomplished in a sprite.

GETTING A HANDLE ON THINGS

To use the higher-order rendering functions (Move(), Draw(), AreaFill(), and so on), graphics.library requires a RastPort structure. After creating a BitMap structure for the sprite, attaching a RastPort is comparatively trivial:

```
struct RastPort rp;

InitRastPort (&rp);
rp.BitMap = spritebm;
```

Like the RastPort attached to an Intuition screen, this RastPort does not protect against rendering into invalid areas. Thus, you must perform all the clipping yourself. If you'd prefer to let the system worry about clipping, you can attach a Layer to the sprite and render through its RastPort. You do so as follows:

```
struct RastPort *rport;
struct Layer_Info *li;
struct Layer  *spritelayer;

if (!(li = NewLayerInfo ()))
  die ("NewLayerInfo() failed.\n");

if (!(spritelayer = CreateUpFrontLayer (li, spritebm,
                    0, 0,
                    15, spr->height - 1,
                    LAYERSIMPLE, NULL)))
                    die ("Create layer failed.\n");

rport = spritelayer->rp;
```

THE EXAMPLE PROGRAM

The program drawsprite in the accompanying disk's Schwab drawer creates a sprite, attaches a RastPort, and renders text into it. The above techniques are collected in the function GetSprRendEnv(), which procures a rendering environment for a sprite.

GetSprRendEnv() accepts a pointer to a SimpleSprite structure and a Boolean. The SimpleSprite structure is used to create the BitMap and RastPort structures. The Boolean indicates whether or not you want a Layer to clip your rendering in the sprite. If so, it will create a Layer and attach it to the sprite. Otherwise, it will create a simple RastPort.

GetSprRendEnv() returns a pointer to a *pointer* to a RastPort. That is, it returns:

```
struct RastPort **
```

You can't use the pointer it returns directly. You must fetch the RastPort pointer out of the pointer it returns, as shown:

```
struct RastPort *rport, **rendenv;

rendenv = GetSprRendEnv (sprite, TRUE);
if (rendenv)
.rport = *rendenv;
```

If GetSprRendEnv() returns NULL, the operation failed, and nothing is allocated.

When you are finished, you should call FreeSprRendEnv() to deallocate the resources procured with GetSprRendEnv(). The routine automatically figures out whether you have a Layer attached and frees it. You pass in precisely the same pointer you got from GetSprRend-

Continued on p. 63

ON DISK

A Presentation Engine For AmigaVision

By John Gerlach, Jr.

WITH AMIGAVISION, COMMODORE redefined the concept of multimedia presentation systems, letting you easily create applications by placing nonintimidating icons on a grid. This ease requires that programmers give up creative control, and that users need to have a *lot* of memory to run the application, right? Wrong. I'll show you some simple ways to reduce your work and the amount of memory needed for even large, complex applications.

AV is a perfect prototyping and development environment for many types of projects, including multimedia applications. The one attribute common to all multimedia applications is the large amount of information required to comprehensively and compellingly convey the topic at hand. Unfortunately, AV is sometimes discredited for its demands on system memory. Versions 1.70 revision Z and above do not use as much memory you might assume they do at first glance. Using AV's newer features to control memory usage allows very complex applications to be run on machines with only one megabyte of memory.

AV1.71Z's disk size of over 590K is not an accurate measure of the amount of memory required when using the program. Overlays curtail memory requirements so the RAM consumed by code is never equal to the space used on disk. In fact, when the executable is started by double-clicking on an application's icon, the maximum memory consumed by AV is approximately 250K (this playback-only configuration may also be initiated from CLI with an –a<filename> specification). CATS is currently controlling the distribution of "player" versions of AV which do not contain any of the editing environment code, thus dramatically lessening the amount of disk space consumed.

DEFAULT CODING MODEL

A negative attribute common to many AV applications is that the programmer was too soon satisfied with what I'll call the default coding model. In this model, the code directly reflects the pattern of movement that the end user will follow, as a "bird's-eye" view of the application would show the default presentation of information. In other words, if the application were to ask a series of questions, the flow would easily exhibit each question's display and user-interactions, in direct succession.

For reference, consider an application that serves as an information kiosk used to query prospective music buyers. The application is to ask each participant questions about their musical preference, price limitations, and other such details. After the customer gives enough information, the kiosk might make selection recommendations, print coupons, and so on.

The general assumptions made for the application are:

1. An "attract loop" is presented while the kiosk is idle to attract the interest of passers-by.
2. The application should allow for many types and formats of questions, making no restrictions on the number of answers available at any time.
3. The response to any question may modify the default movement through the remaining questions. In addition, nonactivity on the user's part forces the attract loop to restart.
4. It must be easy to modify the questions and highlights of the kiosk as dictated by marketing issues.

In the default coding model, the resulting application would be a long and narrow flow that repeats the sequence of displaying a picture, adding hitboxes to let the user make a selection, and determining how to react to the selection. AV's editing environment makes this type of coding as easy as possible by letting you duplicate sections by dragging a copy of an existing icon. You often end up, however, with an application that is difficult to maintain and modify: You have to deal with many similar looking and functioning sections of code and force the logic for every question to constantly reside in memory, even though the customer can only interact with one question at a time.

Even worse than the repetitive coding is the necessity of supporting the third assumption. Yes, AV has two Goto icons, but indiscriminate use of these leads to unmanageable code. Because all questions need to return to the attract loop, you're guaranteed to have multiple branches between parts of the application that cannot be on-screen at the same time.

Being able to easily modify the data presented by the kiosk without worrying about corrupting the basic playback mechanism is paramount for assumption number four. If several questions are added to a musical classification in the middle of the application, and the modification results in an endless loop between only a few questions, your customers will not benefit from the service you are trying to provide.

A PRESENTATION "ENGINE"

Don't give up; you can use this default coding model as a verbose prototype of a smaller and simpler method. To overcome all of the shortcomings of the default model, fragment the application into multiple parts and factor out any similarities. This will lessen the amount of memory required, while potentially easing the burden of authoring, debugging, and supporting the code.

The key to this process is to separate the application's information from the rudimentaries of its operation. With AV's integrated database support, it is easy to have an application rely on information contained in a series of database records. Furthermore, you can let the database control the presentation sequence by adding control information to these records.

Using such a database, the AV application can be transformed into a presentation "engine," which you can construct in a completely general manner for reuse in multiple applications. This distributes the cost of its already short development and allows you to create subsequent applications inside a familiar and previously tested environment.

A presentation engine operates on information found in the database by repeatedly reading the "current" record and performing the action it specifies. This process can been condensed into a loop that performs a database access at the beginning of each iteration. The information contained in each record defines the specifics for the current iteration, but may also contain default settings for the next.

"With AV's integrated database support, it is easy to have an application rely on information contained in a series of database records. Furthermore, you can let the database actually control the presentation sequence . . ."

In the example database, each record contains a default next-record value. If the current iteration's code desires to modify this default (to force a return to the attract loop, for example), it simply overrides the stored value. Otherwise, the movement dictated in the database is followed.

One means of lessening the amount of memory used by a complex application is to use the chaining feature in version 1.70. Chaining allows multiple flows to be connected during runtime, giving you control over the amount of memory consumed by the code. The second assumption in the example forces you to allow plenty of resources for each of the questions. For this reason, only one question should be present in memory at any given time. The code for each question will be loaded and started by the presentation engine only when it is needed. If these flow files are kept small, their loading from disk can easily be masked by other operations.

AV's chaining has two forms: Link mode and Call mode. Link mode starts the new flow and then purges the memory used by the previous code. Call mode treats the new flow as a disk-based subroutine that is loaded and executed, being removed from memory when it terminates. Chaining in both modes is performed without the loss of variables created by the parent. When using Call mode, modifications made by the child to the parent's variables are retained after the child terminates. In Call mode, the child flow can actually create new globals to be used later by the parent. The new function defined() can be used to determine if a variable with a specific name is defined.

To drive the engine, I constructed a sample database containing records with the fields:

- **THISRCRD:** A key field used when selecting a record in the database.
- **NEXTRCRD:** The default next-record number. The value retrieved from the database may be overridden as detailed above.
- **AVFNAME:** The name of an optional .AVf file that is chained to, if specified.
- **SCRNNAME:** The name of an optional ILBM file.

If the current record contains an .AVf filename, the engine loads and executes the file. This flow may wish to modify the next record number or leave the default value in place. If no .AVf filename is specified, the engine simply displays the appropriate picture, delays for two seconds, and continues with the default next record as instructed. This internal/external distinction is completely arbitrary. It is entirely up to you to determine which operations should be coded directly into the engine and which should be placed in individual child flows. Note that it would be simple to create a generic Yes/No child flow that loads the ILBM file specified in the database and uses generic response areas for the user interaction.

The example assumes that "question" 0 is the attract loop, and normally this flow would be as flashy as possible. It is started when the program first runs and after each respondent completes the questionnaire, continuing until the next person signals an interest to participate. At that time, the engine is instructed to present question number one, which begins the query of the participant's opinions. ►

The flow file AttractLoop.AVf (in the accompanying disk's Gerlach drawer) contains the logic behind an ultra-simplistic (boring) attract loop. It also shows how the default next-record value can be modified by a chained flow, resetting it to zero when the engine is not supposed to continue with the presentation. This allows the attract loop to ignore the necessity of coding a looping action. Also note that any of the child flows can use this method of forcing a return to the attract loop without the need for a single Goto icon.

You are now free to make maximum use of the computer's resources, because chaining frees the code for each question as soon as it is completed. An attract loop should make demands on the system's ability to attract attention by showing flashy pictures and making interesting noises, but it probably does not need a lot of analysis coding. On the other hand, a question may depend on previous responses and may lessen its user-interaction demands to allow code space for complicated information analysis.

"An attract loop should make demands on the system's ability to attract attention by showing flashy pictures and making interesting noises, but it probably does not need a lot of analysis coding."

Development of these child flows can occur quite rapidly for several reasons. First, currently loaded flows can be duplicated simply by pressing the key combination Right-ALT-F1 while the window is active. This results in an exact duplicate of the original window that is named Untitled. You can then make modifications and save the flow to disk. This ease in duplication can be used to speed test different approaches to the same issue. Second, when a presentation is started from the editor and chaining is specified, AV searches the files currently open in the editing environment before loading it from disk. This lets you create a test version of the editor without saving to disk and changing the filename in the database.

INTEGRATED DATABASE CREATION

The example described above assumes that development is done in two discrete phases—database creation and application development (and testing). For each iteration during this process, you must enter AV's database editor, load the database file, select and edit one or more records, exit the database editor and reintroduce the flow. This repetition wastes time and can quickly become annoying.

A logical extension to the example is to incorporate the database editing process into the main flow, so that the runtime environment need not be exited when modifying the current data or adding new records. The example flow contains a Key Interrupt icon tied to an IfThen icon that relies on the state of the variable Development. The variable is defined in the first X/Y icon and can be modified to control the availability of the database-editing feature. Set the variable to TRUE during development to make the interrupt available. To enter the database editing section while testing, simply press the F1 key. When beta testing your application, set the variable to FALSE so that users cannot modify the data. Of course, all icons specific to editing can be removed when the project is completed.

EXAMPLE ON DISK

The example in the Gerlach drawer on the disk is divided into three flows. The first, Engine.AVf, is the engine described above, made nongeneric only by the first X/Y icon. This icon controls the activity of the interactive database editing, and the name of the database file and the project's working directory. The remainder of the example relies heavily on variable substitution available through AV's bracket notation. All dependencies on the presentation's directory use the contents of the variable ProjectDir. To move the application to another volume, simply copy the complete directory structure and change the value assigned at the beginning of the flow.

Likewise, the name of the database is assigned to a variable. This is really only useful if some logic is needed to determine which database the presentation engine uses. For instance, before entering the main loop, the flow could ask each user to select an area of interest. This code would then initialize the variable so the engine is controlled by the music database instead of, say, the kitchenware database.

This variable substitution is best exemplified by the icons named "Execute flow as a subroutine" and "Display image." Both show how you can use more than one variable to construct the name of a resource. Specifically, the screen icon shows how powerful one icon can be. Using two variables, it presents any picture supported by AV, without the Intuition pointer, starting with a fade-from-black transition. A series of similarly named pictures could be displayed by adding [counter] to the end of the filename and controlling the variable through a Loop icon.

The second flow file, AttractLoop.AVf, contains the specifics to our default "question." While nothing in this flow is particularly spellbinding, it no longer retains the Module icon given when opening a new flow window. This gives a slight size and speed increase, while reinforcing that Modules need not be the first icon in every flow.

The code in the third flow, DataBaseEditor.AVf, allows interactive creation and editing of database records. Variables used only for editing are defined inside the icon "Create DBaseOP locals" so that they are not allocated until editing is requested, and so that they do not remin after the editing process has ended.

Feel free to use these flow files as patterns for your own presentation engines. ■

John Gerlach, Jr. is Director of Software Development for IMSATT Corp., the creators of AmigaVision. Contact him c/o The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458, or on BIX (jgerlachjr).*

ON DISK

Object-Oriented Display Refreshing

CallLists is a refreshing way to keep your interactive programs up to date.

By Bryan Ford

INTERACTIVE PROGRAMS (containing complex user interfaces) pose problems that you don't have to deal with in batch processing programs. When writing a batch program, such as a compiler, you process a complete set of input data in a certain format and write it out in another format. An interactive program, on the other hand, has to process input data that comes from the user in little pieces at a time (events). These events usually are entered in no particular order, and often change or even reverse the results of previous events. An interactive program must generate new, updated output very soon—no user likes to wait for a slow program—after each input event.

This article deals with one of the main problems in writing interactive programs: refreshing the various parts of a program's display at appropriate times. Some programs have many different windows, possibly with several separate display areas in each window. When some part of the project is changed (a new event is received), the program must immediately update the appropriate windows to reflect the changes.

Probably the simplest method of refreshing displays is to create one big function that calls all other screen refresh functions one by one:

```
void RefreshAll()
  {
     RefreshViews();
     RefreshQuickMenu();
     RefreshThis();
     RefreshThat();
  }
```

This method is very simple, and assures that *everything* is up-to-date after it has been called. While this method works well for simple requesters or very small programs, once you add a more windows or other display areas, the program becomes slow and flickery.

On the opposite extreme, you could call the appropriate refresh functions immediately after each change is input. Although this method can eliminate unnecessary refreshing, it requires you to remember all of the program's interdependencies. It also tends to create subtle bugs and make revision difficult at best.

Two other problems are not adequately addressed in either of these methods. First, what if one refresh function depends on another? For example, in a 3-D modeling program, DrawPolygons() might depend on CalculatePolygons(). These functions must be called in the correct order to ensure that the final display is completely up-to-date—if you draw objects before you recalculate their positions, the display will always remain one step behind the rest of the program. A reliable method must be found to ensure the correct calling sequence.

The second problem arises when a refresh function is called many times quickly. For example, a user might select several menu-based commands at once, each of which performs some simple operation on the project. A poorly written program will often refresh the display after each operation, even though it really only needs to be refreshed once at the end. A more common example is repetition of an operation—how many times have you briefly held down some repeating key in a program and had to wait a long time afterwards for it to finish processing your keystrokes?

These problems can be solved using various combinations of functions and function calls. Using flags to keep track of items that need refreshing can also help. No matter how carefully you lay out the function calls and flags, however, you will still run into problems as the program gets bigger.

A BETTER WAY

A cleaner and more efficient method of handling refreshing is "object-oriented refreshing," which uses data structures and lists rather than direct function calls. As you will see, it can conveniently solve all of the problems I've mentioned using a small amount of extra code. It also helps prevent obscure bugs and makes program revision simpler.

The basic data structure used by the system is as follows:

```
struct CallNode
  {
     struct Node Node;
     LONG (*CallFunction)(LONG GlobalData, LONG LocalData);
     LONG LocalData;
  };
#define CNT_INLIST NT_USER
```

A CallNode represents a function that must be called sometime in the future. Think of it as a "computer agenda item." As you can see, it is based on Exec's standard Node structure and is meant to be linked into a standard Exec MinList (the stripped-down version of the standard List).

When an event occurs causing a particular part of the display to need refreshing, a CallNode is added to a List (a CallList). Instead of calling the refresh function immediately, the call is "remembered" for later. After adding the CallNode, the program can continue processing other events immediately.

A typical program will have one global CallList in the main module and many small CallNodes defined in other modules, each one representing a particular refresh function that might be called at some time. For example: ►

```
static void RefreshQuickMenu(long globaldata,long localdata)
   {
      (do all refreshing of the quick menu)
   }


struct CallNode CallQuickMenu = {
   {0,0,0,<priority>},RefreshQuickMenu,<local data>};
```

Then, wherever the quick menu is changed (or for some other reason needs refreshing), instead of immediately calling RefreshQuickMenu(), just make this call:

```
AddCallNode(&RefreshCallList,&CallQuickMenu)
```

(If a particular CallList is used often, you might want to define a macro that adds CallNodes to a specific CallList.) This adds the CallNode to the specified CallList. The AddCallNode() function is defined as follows:

```
void AddCallNode(struct CallList *l,struct CallNode *n)
   {
      if(n->Node.ln_Type != CNT_INLIST)
         {
            n->Node.ln_Type =CNT_INLIST;
            Enqueue((struct List*)l,&n->Node);
         }
   }
```

When it's time to refresh the display (probably just after all of Intuition's IntuiMessages are handled, but before you call Wait()), you simply make a single call to the function CallRem():

```
long CallRem(struct CallList *l,long GlobalData)
   {
   struct CallNode *n;
   long ret;
      while(n = (struct CallNode*)RemHead((struct List*)l))
      {
         n->Node.ln_Type=0;
         if(ret=(*n->CallFunction)(GlobalData,n->LocalData));
              return(ret);
      }
   return(0);
}
```

This function traverses the entire CallList, removing each CallNode as it progresses, and calls each refresh function in turn. (Note that CallRem(), in the source code in the accompanying disk's Ford drawer, is actually a macro that uses a more powerful version of this function, explained later.) Each CallFunction must return zero if it wants the CallList processing to continue (the usual case) or nonzero if it wants to stop processing the list (if an error occurred, for example).

By now you should have a general idea of how the system operates. The rule of thumb is: Save the refresh function calls until later, and then only call them when the user is no longer doing anything.

MAPPING THE LISTS

The use of the CallNode's ln_Type field prevents any CallNode from being added to a CallList more than once. This helps to prevent corrupting the list, makes the caller's life easier and conveniently solves the problem of queueing up many identical or similar refresh events. You can make as many AddCallNode() calls for a particular CallNode as you want, but the function is only called once at the end, eliminating the need for "dirty" flags and such.

Because Enqueue() is used to add nodes to a CallList, the list always remains sorted according to priority. Because CallRem() starts at the head of the list and works toward the tail, it always calls CallNodes with the highest priorities first to ensure that all refreshing is done in the correct sequence. For example, in our 3-D modeler example, CalculatePolygons() would simply be assigned a higher priority than DrawPolygons() and, therefore, would always be called before DrawPolygons().

As you set up your CallLists and CallNodes, keep track of which functions depend on which other functions, and set the CallNode priorities appropriately. I recommend keeping handy a text file that lists all the CallNodes for a given CallList, their priorities, and where in your source code they are located. This will help you assign priorities to new CallNodes and will give you a good overview of which refresh functions are being called, in which order.

In general, CallLists and CallNodes can simply be defined as global or static variables. I generally define a CallList near the corresponding call to CallRem(), and put CallNodes just below the functions they point to. In some cases, however, dynamically allocating CallNodes may be more appropriate. Just make sure you always allocate the memory with MEMF_CLEAR, or at least initialize the Flags field to zero before using the CallNode. You can also have more than one CallRem() call for a given CallList, if you find this beneficial. Just remember that each call to CallRem() completely empties the list, so the next call won't do anything unless you first add additional CallNodes.

"As you set up your CallLists and CallNodes, keep track of which functions depend on which other functions, and set the CallNode priorities appropriately."

Any CallFunction (a function pointed to by a CallNode and called by CallRem()) may make calls to AddCallNode(), even on the CallList that's currently being traversed. This provides a convenient solution to another dependency problem. In our 3-D modeler example, CalculatePolygons() calls AddCallNode() with the CallNode for DrawPolygons(), forcing a redraw to occur sometime after any recalculation, without either of the functions actually calling the other directly. In general, you should add other CallNodes with lower priority only this way, although it is safe to add any CallNode if you are careful about interdependencies.

MORE SYSTEM FEATURES

For flexibility, the system provides a general-purpose mechanism for passing parameters to the CallFunctions. The GlobalData variable is a LONG (you may use it as a pointer if you wish) that is given to CallRem() and passed through to all of the functions on the CallList, as they are being called. This provides a convenient way to "broadcast" a piece of data to all the called functions. Similarly, each CallNode contains a LocalData variable that is passed to the CallFunction when it is activated. This is particularly useful if you dy-

Continued on p. 63

InovaTools 1 version 2.0

GUI help for both generations

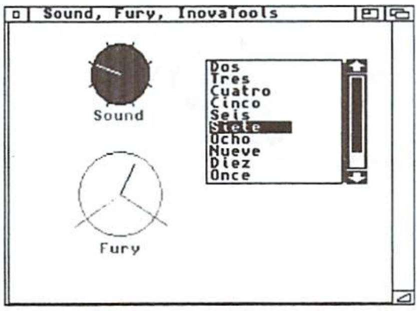
By David T. McClellan

INOVATOOLS IS A package of gadgets and window functions perfect for writing Intuition applications for AmigaDOS 1.3 and, with some work, 2.0. Designed to work with INOVAtronics' PowerWindows, InovaTools provides some nifty wrinkles on PropGadgets, Boolean gadgets, other gadget types, and menus and has some useful window functions. It does not appear to have been updated for OS 2.0, however. The copyright dates on the include and demo files were 1987/1988, and while the demo executables ran fine, I had fun getting the accompanying source working with my compilers. (More on that later.)

The InovaTools routines are Intuition gadgets and utility functions, written to be called and used in the same style as the standard Commodore Intuition gadgets. Briefly, the set is comprised of a general purpose list gadget, a good file requester built on top of the list gadget, knob-style proportional gadgets, drag gadgets (for drag and drop applications), pop-up menus, a color-palette requester, and a number of window, gadget, and utility routines. The package comes with header files and libraries for SAS/C and Manx C, libraries for linking, an AmigaDOS shared library for run-time linkage, an assembly language header/interface, source, and user-contributed link libraries for other languages. I tried the SAS/C (v5.10) and Manx libraries (I used version 5.0a, although Inovatools comes with headers and libraries for v3.4b and 3.6a as well). The most recent SAS compiler the package describes compiling with is v4.0; perhaps this is why I got buggy executables using 5.10 SAS/C.

THE GADGETS

Inovatools has two gadgets not seen



in Intuition 1.3 or 2.0—Drag and Knob gadgets. Drag gadgets are Boolean gadgets that can be picked up and dragged around. Knob gadgets are an alternate form of the slider-style PropGadgets we all know and love, and are useful for situations in which a slider looks wrong. They can rotate freely or in fixed steps and can have minimum and maximum angles of rotation.

To use a DragGadget, you first lay out a DragInfo data structure:

```
struct DragInfo {
   struct Gadget *Gadget;    /* Ptr to
      BOOLGADGET */
   SHORT LeftEdge, TopEdge, RightEdge,
   BottomEdge;
   USHORT Flags;
   SHORT XPos, YPOS;
   void (*UpdateRoutine) (); /* User supplied */
   LONG DD;           /* Don't touch */
};
```

As you can see, it looks like most Intuition data structures; INOVAtronics designed the InovaTools structures to fit easily into your programs.

The Gadget pointer at the beginning of this struct points to a BOOLGADGET, which would be described elsewhere in your program and would be linked into your Window's gadget list.

LeftEdge through BottomEdge define a bounding box for your DragGadget; use normal numbers within the window's bounds to constrain the gadget to an area within the window. If you want it to be free to move anywhere, set LeftEdge and TopEdge to large negative numbers, such as –32767, and set RightEdge and BottomEdge to positive numbers greater than any possible window bound.

The Flags field specifies further options for the DragGadget: DRAG_INWINDOW, DRAG_MOVEGADGET, and DRAG_OUTLINE. DRAG_INWINDOW constrains the gadget to stay within its window whatever its bounding box is; otherwise it can be moved to any window. DRAG_MOVEGADGET causes the gadget to stay where it's released instead of snapping back, and DRAG_OUTLINE instructs the drawing routine to treat color 0 of the gadget's image as transparent (for non-rectangular images).

The UpdateRoutine field points to a call-back routine you provide, which is called whenever the gadget changes position after you hand control to the InovaTools' drawing routine. You can use it to change imagery or update your own data structures.

XPos and YPOS contain the gadget's position at any time, such as when your call-back routine is called.

DD is private to Inovatools.

The next step is to initialize your windows, Boolean gadgets, and others. When your message loop receives a GADGETDOWN message for the BOOLGADGET, call DragGadget() with a pointer to the Window containing the DragGadget and a pointer to the DragInfo struct, which should point to the BOOLGADGET. DragGadget() returns when the user releases the DragGadget and inspects its XPos and YPOS fields at that time to find out where the gadget landed. The WhichWindow() routine tells your program which other window the DragGadget landed in if it didn't have its DRAG_INWINDOW flag set; this is the kind of thing you want for drag-and-drop applications such as dragging records to file cabinets, files to shredders or printers, and so on.

The Knob gadget must be rendered into your Intuition screen after your OpenWindow() call, but otherwise has a similar use model. When your application gets a SELECTDOWN MOUSEBUTTON event, it calls an InovaTools routine to handle the event. If the SELECTDOWN happened over a Knob, the KnobGadgets() routine ►

processes all further input and Knob rendering until it receives a SELECT-UP and returns a pointer to the Knob. Otherwise it returns NULL, and your application can further process the SELECTDOWN on its own. As with DragGadgets(), you can provide an update routine that InovaTools calls each time the Knob rotates. This allows the Knob rotation to visually or audibly affect other parts of the application, such as sound volume, image size, or color levels.

REQUESTERS

InovaTools comes with a generic list requester, a good file requester built on top of the list requester, and a color-palette requester built with Knob gadgets instead of sliders. In addition, it provides a cross between requesters and menus—popup menus, which appear after the left-mouse-button event you specify. The generic list requester was obviously built prior to 2.0 Intuition. Under 1.3 it would have been very useful, but now most applications prefer 2.0-style lists.

The list requester has four gadgets that pass events to your program: List, ClickUp, ClickDown, and Scroll. These respectively cause events when the user selects an item in the list, clicks the Up or Down arrow, or slides the scroll bar "thumb." This requester also calls an application-provided routine to draw each list entry; this routine is called when the list is initially drawn, once for each item in the visible list, and then each time an item scrolls into view. InovaTools passes the routine an x,y coordinate to draw, a pointer to the List item to draw, and a highlighting flag. To make things easy, only the first field of the List item struct need be a pointer to the next item. The List handler walks the singly-linked list of List items you provide it, rendering the relevant parts of the list, and scrolling when you request. When the the user clicks on the List gadget, your program is signaled that the user picked a list item. Your program can eliminate the list or use that choice in combination with other gadgets built into your own requester (as the file Requester does). As a plus, InovaTools provides routines to insert and delete items into and from the list, scroll the list up or down by one or more items, resize the list, close the list, and determine position information about list items.

I've used several public domain pop-up menus (menus that pop-up at the mouse position) and the InovaTools Pop-Ups are as useful as the best of these. Basically, at an event such as a GADGETDOWN or SE-LECT-DOWN, your program calls the PopUpMenu() routine with a standard Intuition menu struct, which can include submenus. PopUpMenu() displays one bar and side submenus pulled off it; it does not display or affect the traditional menu bar. PopUpMenu() returns a standard menu selection code that your program decodes with the ITEMNUM and SUBNUM Intuition macros to get the user's choice (including none, if they deselected the menu). These are very easy to use and handy in applications in which the user often does not want to move the mouse up to a menu bar to choose an action.

The two composite requesters—file and color-palette—are pretty standard, except that the color-palette uses Knobs to handle RGB and HSV colors. The file requester supports standard directory and device searching, can be set to look for files with a certain extension (not a complete wildcard), and runs under any Intuition screen. The color-palette requester has six knobs, three for RGB and three for hue, saturation, and intensity (value), and will also run in any Intuition screen. I found the file requester fairly useful, especially for cases where you don't want full wildcarding; you call it and it gives you back a filename. I found the color-palette editor less useful. Perhaps I'm too used to sliders, but the RGB knobs were more awkward for fractional control.

In addition, InovaTools provides several handy utility routines to deal with gadgets, windows, and menus reentrantly—to enable and set components of gadgets by window and ID rather than by pointer, to clone a window and its gadget list, and to deal with menu components by ID for checking and enabling. Being able to clone a window and its gadget list is handy for building modeless requesters and multiwindowed applications, and the gadget-by-ID routine is a prerequisite for doing this. It's also a useful abstraction, one I prefer over the gadget-by-pointer method (yes, I do a lot of MS-Windows programming too). Lastly, Inovatools also has "flashy" window open and close routines, which cause the opening window to rapidly expand from a point to full size and collapse back at close.

PROGRAMMING

Inovatools comes with SAS and Manx linkable libraries and an AmigaDOS library that you can attach at runtime with OpenLibrary() as a alternative (the library must be installed on user's Amigas, in this last case.) Also included are beta-level libraries and headers for Benchmark Modula-II, TrueBasic, and MultiForth. Try them, but they're not guaranteed. I had problems with both the SAS and Manx versions, just trying to link the demo application. The Manx version first linked with some undefined symbols that, I eventually discovered, came from the floating-point library and were probably used by the Knob gadget; the routines were called by the Inovatools library. The SAS version linked right away, but crashed off and on in the cloned-window and Drag-gadget routines. The file requester, Knob gadget, and list requester all worked fine, as did the executables of the supplied demos. The problem turned out to be some pointer and structure offsets that the routines weren't expecting; I'm still tuning it for DragGadgets. Lists and Knobs appear to work satisfactorily, and pop-up menus as well.

The manual is pretty good for a developer-oriented package like this; each routine and data structure is well described, and source code for all the routines is included.

For two reasons, I can't wholeheartedly recommend this package. I like what the author has done, and the style and docs are good. But, some of the features are outdated (the list requester for one), and the routines don't work right out of the box with all compilers. You have to tune and tweak InovaTools' and your compiler options. InovaTools is useful (more so for 1.3 Intuition applications), yes, but you must "tinker under the hood" to the get full effect. ■

InovaTools 1 version 2.0
INOVAtronics
8499 Greenville Ave., Suite 209B
Dallas, TX 75231
214/340-4991
$99.95
No special requirements.

TNT

Technical News and Tools from the Amiga Community.

Compiled by Linda Barrett Laflamme

Storm Watch

Plugging into your A500 or A2000 processor socket, **Stormbringer** sports 68030 and 68882 chips and up to eight megabytes of RAM. The Stormbringer accelerator is available in several speeds; the 16 MHz board is $1099, the 24 MHz model is $1199, and the 30 MHz unit is $1549. The two top-of-the-line models are a $2349 board with a 50 MHz 030 and a 60 MHz 68882, and Stormbringer 55sync ($2599) with a synchronous 55 MHz processor and coprocessor, which promises to run 35% faster than the standard 50 MHz version. (Prices include 4MB ZIP-RAM.) The board's disable switch comes in handy for finicky programs. Also included in the package is software that activates burst mode, caches, and loads Kickstart into RAM. Stormbringer's German developer Memphis Computer Products GmbH does not as yet have a U.S. distributor, but you can contact the company directly at Gartenstr. 11, 6365 Rodheim v.d.Hohe, 0049-06007/7789,8690.

A New Team of Old Friends

What do you do when you have two popular products that both require the CPU socket? Combine them into an all-in-one solution is ICD's answer. The **AdSpeed/IDE** pairs AdSpeed, ICD's 14 MHz accelerator for the A500 and A2000, with AdIDE, the company's A500/A2000 interface for IDE hard drives. The benefits to you are a 68000 CPU that runs at twice the normal clock rate, 32K of high-speed static RAM for a 16K cache and 16K of cache tags, a 7 MHz mode for compatibility (selectable via the accompanying software or an optional hardware switch), and an autoconfig, auto-booting interface for IDE hard drives. The combo is available in three packages: the AdSpeed/IDE-40 for installing an IDE drive in an A2000, the AdSpeed/IDE-44 Kit for installing a 2.5-inch IDE drive in an A500, and the AdSpeed/IDE-40 Kit for replacing an A500's internal floppy with a hard drive. For prices, contact ICD Inc., 1220 Rock St., Rockford, IL 61101, 815/968-2228.

C Scoop

If you're struggling to learn the basics of SAS/C, consider consulting ***Observations***, SAS Institute's new technical journal. Published quarterly in a joint effort of the SAS technical support and documentation development departments, *Observations* promises to cover technical applications and techniques for the company's programming tools and operating systems interfaces, as well as statistics, graphics, and display software. Issues are $15 apiece or $49 for a year's subscription. For more information, contact SAS Institute's book sales department at SAS Campus Dr., Cary, NC 27513, 919/677-8000.

Organized and Backed-Up

Tired of clicking to the seventh level of your hard drive's directories just to run one program?

Display Systems International's **Hard Disk Organizer** ($44.95) lets you run applications and scripts with only one click. Each of the unlimited number of buttons can be programmed with up to eight AmigaDOS commands, so you can include any assigns, directory changes, and so on necessary to run your applications. To help you group your applications by type and quickly identify them, each button can be assigned one of ten colors. Direct your inquiries to Display Systems International, 147 West Main St., Dayton, PA

Continued on p. 34

STANDARD AMIGA

At last, a development standard for the next generation of Amiga Graphics and Video Software.

For the ultimate in high-resolution 8 to 32 bit graphics and video applications, Digital Micronics and Progressive Peripherals & Software jointly developed a communications interface for the Texas Instruments 340x0 high performance graphics processor family and the Amiga's 680x0.

Applications developed with *SAGE* will run on DMI's Resolver™ Graphics Board, PP&S's Rambrandt™ Video/Graphics board, or any other video/graphics board which supports the *SAGE* instruction set - NTSC and PAL systems.

Focus Your Creativity, and Reach New Markets.

With *SAGE*, it's easy to quickly port existing Amiga software, or develop new high-performance applications. *SAGE*'s software system provides a simple, efficient way to send instructions and data back and forth between the Amiga and *SAGE*-compliant graphics board(s). *SAGE* is based on Texas Instrument's TIGA interface specification, the leading graphics standard for PC-compatible systems. *SAGE* goes beyond this specification by incorporating many Amiga specific enhancements.

The Right Tool for the Job.

The core *SAGE* library consists of over 120 optimized, high-performance functions. Geometric drawing functions include: line, oval, ovalarc, piearc, point, polygon, polyline, rect(angle), seed, and more. Style functions include: draw, styled, pen, pat(ter)npen, fill, patnfill, frame, patnframe, and more. Core system functions include: init_cursor, set_cursor_xy, set_ppop, set_transp, and many more.

SAGE functions provide easy access to the many powerful graphic capabilities of the TI 340x0 processors. There are functions for setting transparency, defining how pixels are combined, bit-blitting, text definition and display, hardware clipping, page-flipping animation, and many more. *SAGE* functions provide the resources which graphic and video application developers need the most, from low-level to high-level graphics and video processes. The *SAGE* library documentation thoroughly details the use of all *SAGE* functions.

Power for Present and Future Applications.

SAGE insures expandability by allowing you to create your own custom libraries.

The functions provided are only the beginr you can create your own executable functions to streamline and optimize your software. Future updates planned for the *SAGE* library include JPEG decompression for real-time video applications, and enhanced 3-D functions libraries.

SAGE extensions for DMI's Resolver™ take full advantage of the Resolver™'s ultra high-resolution 1280 x 1024 double buffered display and onboard TI 34010 60MHz processor.

PP&S provides additional *SAGE* libraries for use with Rambrandt™. With these libraries, multiple board/multiple monitor systems can be configured in a virtual desktop system, to distribute a single image over multiple monitors. Other Rambrandt™ Library functions support parallel processing, alpha channel functions for video overlay, transitions between buffers ("A/B rolls"), fast image loading, multiple 256-color images on the same screen, and 3-D graphics/TI 34082 coprocessor functions for 3-D specific modeling and rendering.

Many respected, successful Amiga developers have pledged their support to *SAGE*. For information on how to receive the *SAGE* development system, contact DMI or PP&S.

GRAPHIC EXTENSION

SAGE

Standard Amiga Graphic Extension

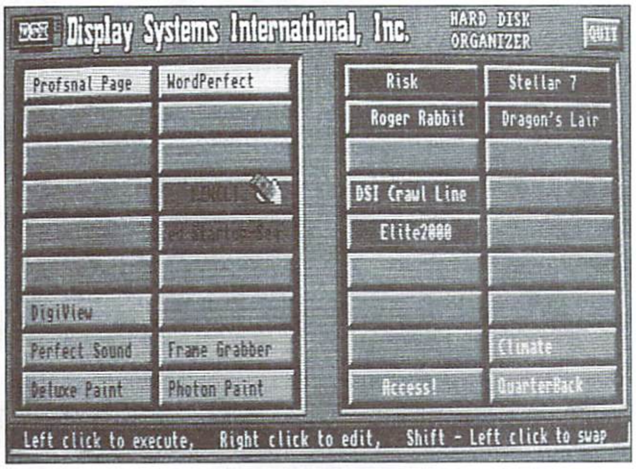
PROGRESSIVE PERIPHERALS & SOFTWARE
Attn: Steve Spring
464 Kalamath Street
Denver, CO 80204-5020
Tel: (303) 825-4144
Fax: (303) 893-6938

DIGITAL MICRONICS, INC.
Attn: Dean Agar
5674-P El Camino Real
Carlsbad, CA 92008-7130
Tel: (619) 931-8554
Fax: (619) 931-8516

From p. 31

16222, 814/257-8210.

For more power and flexibility, consider **Directory Opus** ($59.95). The program offers configurable menus, user buttons (84), and volume/device-name buttons (24); the ability to load or edit a config file, on-line help, unlimited directory history, a full directory tree, DOS error code help, ARexx support, CPU-usage and memory monitors, and a date/time display. Directory Opus plays ANIMs, ANIMBrushes, 8SVX sounds, raw data sounds, and SoundTracker files; shows brushes, pictures, and icons; displays fonts and Hex, and runs executables (with user-definable parameters) and CanDo decks; all you do is double click on the file's name. For more information, contact INOVAtronics, 8499 Greenville Ave. #309B, Dallas, TX 75231, 214/340-4991.



Simply click to execute applications and scripts from Hard Disk Organizer.

Once you have your hard disk efficiently organized, back it up! To help, Central Coast Software recently upgraded its popular back-up utility. **Quarterback 5.0** boasts such new features as integrated streaming tape backup, compression, optional password protection and encryption, ARexx and 2.0 support, and additional backup and restore options. The company also promises performance, the user interface, and file-selection versatility were improved. Plus, you can now backup or restore to up to four floppy drives. For $75, Quarterback 5.0 is available from Central Coast Software, PO Box 164287, Austin, TX 78716, 512/328-6650.

Development Watchdog

Designed for the Amiga and composed of 11 command-line utilities, **QVCS** (Quma Version Control System) automatically tracks your source and binary files through development, saving the differences between various revisions in a single file. The lock-file protection lets you use multitasking QVCS on networked systems, and for multi-developer projects QVCS will coordinate accesses and updates to modules that are shared by several programmers.

Need to backtrack? The program retrieves previous file versions and supports journaling. With this feature on, QVCS logs all file changes into a protected journal file, supplying the name of the file modified, the time and date, the name of the person changing the file, and a synopsis of the command used.

Other handy features include an editable message file that defines the message strings used by QVCS utilities, a file-compare utility for both binary and ASCII files, accidental deletion protection, and the ability to automatically expand such keywords as $Revision$, $Author$, $Date$, Log, $Version$, and VER.

To acquire QVCS you'll need $99, to run it, one meg of RAM. For all the details, contact Quma Software, 20 Warren Manor Court, Cockeysville, MD 21030, 410/666-5922.

What's on the Schedule?

If you or your company has a hot new product on the way, tell us about it, and we'll tell the readers. Send your press releases and announcements to *The AmigaWorld Tech Journal,* 80 Elm St., Peterborough, NH 03458, or llaflamme on BIX. ■

The AmigaWorld Tech Journal Disk

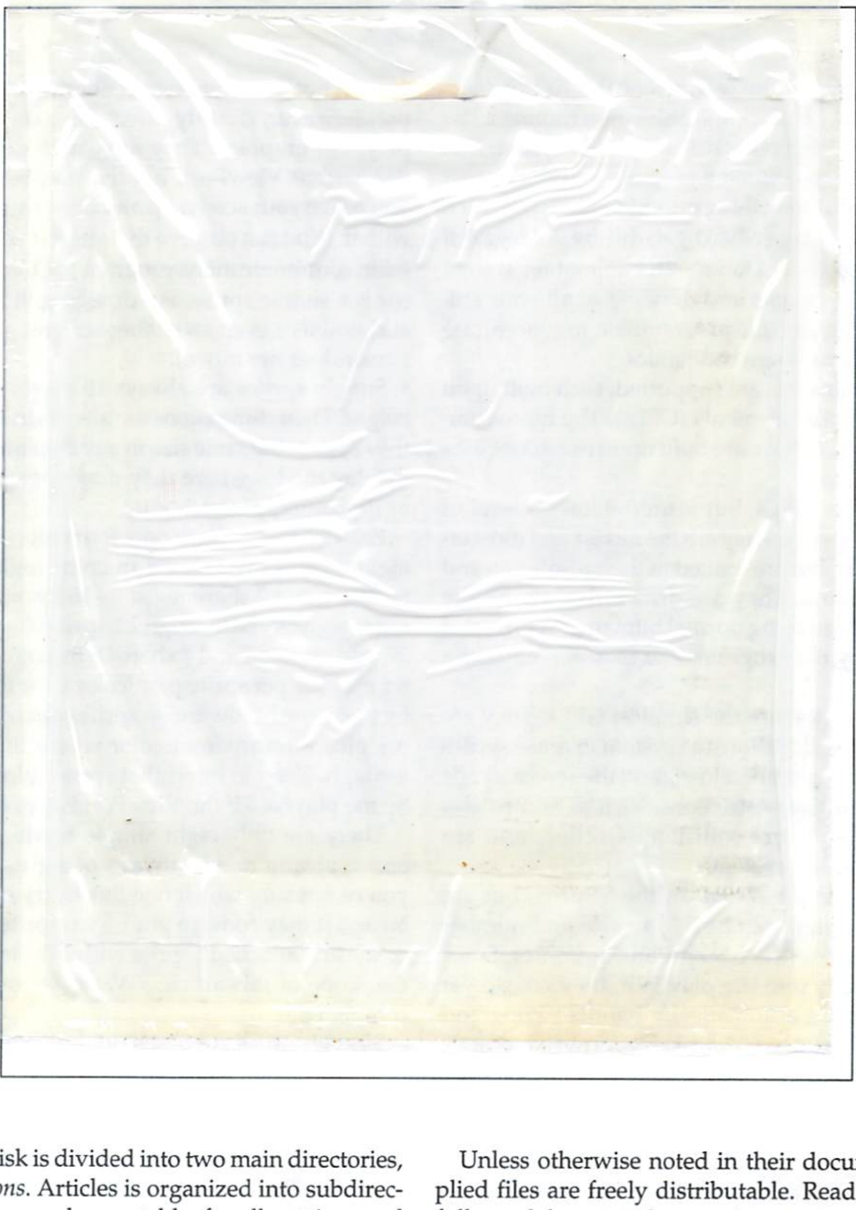
ON DISK

You'll find source code and executables for the article examples, including:

Revision Control System—A program development tracker

Animation Routines—From simple to hierarchical motion

SetRemind—A reminder utility in ARexx



This nonbootable disk is divided into two main directories, *Articles* and *Applications*. Articles is organized into subdirectories containing source and executable for all routines and programs discussed in this issue's articles. Rather than condense article titles into cryptic icon names, we named the subdirectories after their associated authors. So, if you want the listing for "101 Methods of Bubble Sorting in BASIC," by Chuck Nicholas, just look for Nicholas, not 101MOBSIB. The remainder of the disk, Applications, is composed of directories containing various programs we thought you'd find helpful. Keep your copies of Arc, Lharc, and Zoo handy; space constraints may have forced us to compress a few files.

Unless otherwise noted in their documentation, the supplied files are freely distributable. Read the fine print carefully, and do not under any circumstances resell them. Do be polite and appreciative: Send the authors shareware contributions if they request it and you like their programs.

Before you rush to your Amiga and pop your disk in, make a copy and store the original in a safe place. Listings provided on-disk are a boon until the disk gets corrupted. Please take a minute now to save yourself hours of frustration later.

If your disk is defective, return to AmigaWorld Tech Journal Disk, Special Products, 80 Elm St., Peterborough, NH 03458 for a replacement.

ON DISK

Programming Motion: Animation Elements

By Paul Miller

ONE OF THE many unique features of the Amiga is its built-in set of low-level graphics and animation routines. The graphics library has a very powerful animation system that allows you to set up your animated graphics and then forget about them, at least until something important happens (such as, say, your animated super-hero gets hit by a MegaKill Bomb or falls into the Pit of Doom). The animation system keeps track of the movements and drawing of all your animated objects, allowing you, the programmer, to concentrate on collision and other program mechanics.

Several levels of animation are supported, each built upon different types of Graphic Elements (GELs). The more complex Graphic Elements, in turn, are built upon previous GELs in a hierarchical manner.

Simple Sprites are not GELs, but form the lowest level of moveable graphics support. They are the fastest and most efficient graphics objects, but are limited in their resolution and restricted to eight colors. They are drawn directly by the graphics hardware, overlaying normal bitmap graphics, and must be controlled by the program. The mouse pointer is a hardware sprite.

Virtual Sprites (VSprites) are the simplest GELs. They are based on simple sprites, but allow the system to reuse sprites as needed. More than eight are allowed on the screen at one time, with certain position restrictions. Virtual Sprites also contain support for software collision detection, and are drawn for you by the GEL system.

Blitter Objects (BOBs) are based on the VSprite, but are handled by the Blitter and can be of any size and number of colors with respect to the background graphics. Since they are drawn directly into the playfield, they are slower than VSprites. They can automatically handle saving and restoring of the background, double-buffering, and software collision detection. They form the basis of animation components.

Animation Components (AnimComps) are combined to make up a complete animation object, each part representing one frame of one piece of the entire object. Each AnimComp is attached to a BOB which contains that frame's imagery.

Animation Objects (AnimObs) are complete objects handled directly by the animation system. They can be composed of many linked AnimComps and support several types of animation.

THE HARDWARE SPRITE SYSTEM

Simple Sprites are drawn directly by the display hardware and are overlayed on playfield (bit-mapped) graphics. (They can be coaxed into being rendered under certain playfield planes, but in common practice this is not done.) Sprites do not, however, directly affect the data which makes up the playfield graphics. They are also rendered with respect to the current ViewPort, not the playfield graphics, so when you move your screen up and down, the sprites do not move with it. (You can observe this affect if you have the public domain communications program JRCOMM. The blinking cursor is a simple sprite. Try dragging the JRCOMM screen up and down or even switching screens, and you'll see that the cursor does not move).

Simple sprites are always 16 pixels wide and can be any height. Their dimensions are always in low-resolution pixels; they appear the same size in any display mode (except 31KHz display modes where they may appear vertically squashed or horizontally stretched).

Each sprite can have only four colors, and the colors for all eight sprites are located in color registers 16 through 31. Sprites 1 and 2 share registers 16 through 19, sprites 3 and 4 share registers 20 through 23, sprites 5 and 6 share 24 through 27, and sprites 7 and 8 share 28 through 31. The lowest register number per sprite pair (color 0 for that sprite) is not used by the sprite hardware, so sprites can really be only three colors plus a transparent color where the background shows through. Keep in mind that sprite color registers are shared by the playfield if the ViewPort is five or more planes deep.

There are only eight simple hardware sprites available, and typically one is always being used by the system (if you're not sure which one this is, try moving your mouse a bit and it may come to you). Two sprites can be combined to create an "attached" sprite with 16 colors, but this is beyond the scope of this article. (Watch for more details in an upcoming issue.)

Simple sprites are based on the SimpleSprite structure:

```
struct SimpleSprite
{
   UWORD *posctldata;
   UWORD height;
   UWORD x, y;
   UWORD num;
};
```

Simple sprites are handled with four simple functions found in the graphics library. The GetSprite() function allocates a new sprite from the sprite system.

```
spritenum = GetSprite(sprite, num);
```

Here, sprite is a pointer to a SimpleSprite structure, and num is the sprite number (0–7) requested. If num is –1, Get-

Harness the power of the Amiga's built-in Sprites, VSprites, BOBs, AnimComps, and AnimObs.

Sprite() will attempt to allocate the first available sprite. Spritenum is the number (0–7) of the allocated sprite, if one was available, or –1 if no sprites were available.

```
struct SimpleSprite sprite;
WORD spritenum;

/* set up the initial sprite information */
sprite.Height = 8;
sprite.x = 0;
sprite.y = 0;

spritenum = GetSprite(&sprite, –1);     /* any sprite */
if (spritenum == –1)
{
/* exit */
}
```

The ChangeSprite() function is used to change the appearance of a sprite.

```
ChangeSprite(vp, sprite, data);
```

Vp is a pointer to the ViewPort that the sprite is based in, or to NULL if the sprite is in the current View. Sprite is a pointer to a SimpleSprite allocated with GetSprite(). Data is a pointer to a block of memory describing the new sprite imagery, as well as some additional control information. The format of this data is as follows, and it must reside in chip memory:

```
struct spriteimage {
   UWORD posctl[2];
   UWORD data[height][2];
   UWORD reserved[2];
};

/* sprite data for an 8x8 checkered square */
UWORD chip s_data[] = {
   0, 0,          /* position/control */
   0xF000, 0x0F00,
   0xF000, 0x0F00,
   0xF000, 0x0F00,
   0xF000, 0x0F00,
   0x0F00, 0xF000,
   0x0F00, 0xF000,
   0x0F00, 0xF000,
   0x0F00, 0xF000,
   0, 0          /* reserved */
};
```

```
struct SimpleSprite sprite;
ChangeSprite(NULL, &sprite, s_data);
```

The MoveSprite() function changes a sprite's position.

```
MoveSprite(vp, sprite, x, y);
```

Vp is a pointer to the ViewPort that the sprite is based in, or NULL if the sprite is in the current View. Sprite is a pointer to a SimpleSprite structure allocated with GetSprite(). X and y are the coordinates where the sprite should be moved to, in ViewPort pixels. (If vp is NULL, the coordinates are assumed to be in low-resolution pixels). Pixel precision for sprites is always low resolution, however, even if your ViewPort is hi-res.

```
struct SimpleSprite sprite;

MoveSprite(NULL, &sprite, 160, 100);
```

When you're finished with the sprite, use the FreeSprite() function to release it.

```
FreeSprite(spritenum);
```

Spritenum is the sprite number to free.

```
struct SimpleSprite sprite;
WORD spritenum;

spritenum = GetSprite(&sprite, –1);

/* error checking - use sprite */

FreeSprite(spritenum);
```

To use sprites in an Intuition screen, be sure to set the SPRITES flag when opening the screen. All sprites can be turned on and off at once with the macros ON_SPRITE and OFF_SPRITE, defined in gfxmacros.h. These actually enable and disabled sprite DMA at the hardware level.

See the program Sprite on the disk for complete example code on using SimpleSprites.

VIRTUAL SPRITES

Virtual sprites are the simplest form of GELs, and are handled by the animation system. The animation system converts a virtual sprite into a simple sprite for rendering, so virtual sprites have the same restrictions as simple sprites, except that there can be more than eight virtual sprites, as long as there are no more than eight on a single scanline. Virtual sprites also have built-in collision detection and handling, ►

and can be extended to include user information for control purposes.

Virtual sprites are described by the VSprite structure:

```
struct VSprite {
    struct VSprite *NextVSprite;
    struct VSprite *PrevVSprite;
    struct VSprite *DrawPath;
    struct VSprite *ClearPath;
    WORD OldY, OldX;
    WORD Flags;
    WORD Y, X;
    WORD Height;
    WORD Width;
    WORD Depth;
    WORD MeMask;
    WORD HitMask;
    WORD *ImageData;
    WORD *BorderLine;
    WORD *CollMask;
    WORD *SprColors;
    struct Bob *VSBob;
    BYTE PlanePick;
    BYTE PlaneOnOff;
    VUserStuff VUserExt;
};
```

This structure contains all necessary hooks for linking the VSprite into the graphic element list, collision handling, and AnimComp control when linked to a BOB.

The width of a VSprite is in WORDs, and for a true VSprite (not linked to a BOB), it is always 1. The depth is always 2 (because sprites can have only four colors). These fields can be different when using BOBs. The position of a VSprite is set by changing the X and Y fields of the VSprite structure.

```
vsprite.Width = 1;    /* always for true VSprites */
vsprite.Depth = 2;    /* ditto */

vsprite.X = 0;
vsprite.Y = 0;
```

Because this is a true VSprite (not being used as a BOB), the VSPRITE flag must be set in the VSprite's Flags field.

```
vsprite.Flags = VSPRITE;
```

The ImageData pointer points to the sprite imagery, which must reside in chip memory. This data is organized the same way as for SimpleSprite imagery, without the extra control information. Each line of data is described by two 16-bit WORDs, so the entire VSprite image is stored in Height*2 WORDs. The first WORD contains the least-significant bit of the color select mask for each pixel, and the second WORD contains the most significant bit. These bits are combined to form a binary color select mask as described below:

```
00 — "transparent" color (unused register)
01 — VSprite's first color
10 — second color
11 — third color
```

```
/* data for a checkered square 8x8 pixels */
UWORD vs_data[] = {
    0xF000, 0x0F00,
    0xF000, 0x0F00,
    0xF000, 0x0F00,
    0xF000, 0x0F00,
    0x0F00, 0xF000,
    0x0F00, 0xF000,
    0x0F00, 0xF000,
    0x0F00, 0xF000,
};
vsprite.Height = 8;
vsprite.ImageData = vs_data;
```

Each VSprite can have its own set of three colors, which will be used when the VSprite is drawn. In this case, the SprColors field points to an array of three 16-bit color values.

```
UWORD vs_colors[] = {0x0F00, 0x0FFF, 0x000F};
/* red, white, and blue */

vsprite.SprColors = vs_colors;
```

If SprColors is NULL, the VSprite will be rendered in the colors already residing in the playfield's colormap at the registers used by the VSprite. Because the system will dynamically allocate a SimpleSprite for the VSprite when it is drawn, you have no way of knowing which colors will be used to render your VSprite, so it is best to specify a color array for each of your VSprites.

Note that the colors used to draw your VSprite will be loaded into the display's color table when the VSprite is drawn, so your background playfield may change colors if parts of it are drawn using the same registers. To avoid this problem, you can use a four-bitplane display, which only uses the lower 16 colors, avoiding the colors used by the sprite system. Or, you can avoid using colors 17–19, 21–23, 25–27, and 29–31 to draw your playfield graphics. If two VSprites have the same colors, you can assign both SprColors fields to the same color array, and the system will attempt to pair the two VSprites, thus helping to avoid color conflicts.

USING VSPRITES

VSprites are added to the current display's GEL list (which resides inside the display's RastPort) with the AddVSprite() function.

```
struct RastPort rastport;
struct VSprite vsprite;

AddVSprite(&vsprite, &rastport);
```

The VSprite must be initialized correctly before it can be added to the GEL list or unpredictable and potentially ugly things may happen to your display.

To remove a VSprite from the display, use the RemVSprite() function.

```
RemVSprite(&vsprite);
```

Make sure that the VSprite has been added with AddVSprite() before attempting to remove it.

Once all the VSprites have been added, they must be sorted so the system can assign simple sprites in the proper order. Use the SortGList() function for this.

```
SortGList(&rastport);
```

To instruct the system to actually render the VSprites, call the DrawGList() function.

```
struct ViewPort viewport;
struct RastPort rastport;

DrawGList(&rastport, &viewport);
```

You must use SortGList() before calling DrawGList() when VSprite positions have changed, as the sorting step is essential for proper allocation of simple sprites.

Now you must tell the system that display imagery has changed. If you're using Intuition, call MakeScreen() and RethinkDisplay(). If you're handling the display yourself, perform a MrgCop() and LoadView(). Do a WaitTOF() first to synchronize the display of the VSprites with the vertical-blanking period. MrgCop(), however, can be quite slow (relatively speaking), so it may be more appropriate to put your WaitTOF() just before the LoadView(). The example used here is based on information from the ROM Kernel manuals, so it remains intact for "correctness" sake.

```
/* under intuition */
struct Screen *screen;

SortGList(&screen->RastPort);
DrawGList(&screen->RastPort, &screen->ViewPort);
MakeScreen(screen);
RethinkDisplay();          /* calls WaitTOF() for you */


/* custom display */
struct View view;
struct ViewPort viewport;
struct RastPort rastport;

SortGList(&rastport);
DrawGList(&rastport, &viewport);
WaitTOF();
MrgCop(&view);
LoadView(&view);
```

VSprites are always drawn in sorted order along the Y and then the X axis, starting at the upper-left corner of the screen. Later VSprites will overlap VSprites drawn above and to the left of them.

Between calls to DrawGList(), the position of a VSprite can be changed dynamically by altering the X and Y fields, and the imagery can be changed by pointing the ImageData field to new image data.

Before any of these GEL functions may be used, the GELs system must first be set up properly (this must be done for each RastPort that GELs will be used in). To do this, you need to properly initialize a GelsInfo structure and several areas of memory used by the GELs system, as well as two dummy VSprites. Then InitGels() must be called with this information.

```
struct GelsInfo gelsinfo = {0};
struct VSprite vs_head = {0};
struct VSprite vs_tail = {0};
struct RastPort rastport;

/* initialize GelsInfo data */
InitGels(&vs_head, &vs_tail, &gelsinfo);
```

See the VSprite program in the Miller drawer on the disk for complete VSprite example code, as well as a generic GEL system initialization routine, to be used before any of the GEL functions can take place.

WHAT ABOUT BOBS?

BOBs (Blitter OBjects) form a more powerful animation subsystem that can be used to create animated objects capable of complex hierarchical movement.

Although BOBs have VSprites at their cores, they are much more flexible. A BOB can be as large as you like, memory permitting, and can have as many colors as the playfield. Think of it as essentially a standard BitMap graphic with GEL information.

Because BOBs are actually drawn into the display playfield, they are somewhat slower than VSprites, and extra precautions must be taken to avoid trashing background imagery. These precautions are, however, handled mercifully by the animation system.

With a few exceptions, BOBs share all the standard GEL information with VSprites (this is logical, since a BOB is essentially an extended VSprite).

The location of a BOB (X and Y) is in pixel coordinates with the same resolution as the playfield. The size of a BOB is also in background pixels, and the width may be more than one WORD. The Flags field does not contain the VSPRITE flag, and the SAVEBACK and OVERLAY flags may be used for additional features. The depth of a BOB may be as deep as the background playfield. BOB image data is in a different format than VSprites, but is still pointed to ImageData. SprColors should be NULL, and the VSBob field points to a Bob structure.

```
struct Bob {
   WORD Flags;
   WORD *SaveBuffer;
   WORD *ImageShadow;
   struct Bob *Before;
   struct Bob *After;
   struct VSprite *BobVSprite;
   struct AnimComp *BobComp;
   struct DBufPacket *DBuffer;
   BUserStuff BUserExt;
};

struct VSprite vsprite = {0};
struct Bob bob = {0};
UWORD bob_image[] = {
   /* BOB image data, organized in planes like a BitMap.
   In this case, there will be 2*32 (64) WORDs per plane,
   times 5 planes, so there will be 640 bytes of image
   data total */
};

vsprite.X = 0;
vsprite.Y = 0;
vsprite.Width = 2;             /* up to 32 pixels wide */
vsprite.Height = 32;
vsprite.Depth = 5;             /* up to 32 colors in BOB */
vsprite.Flags = NULL;          /* no special features */
vsprite.SprColors = NULL;
vsprite.ImageData = bob_image;
vsprite.VSBob = &bob;
```

The BOB itself must be linked to the VSprite as well, so the BobVSprite field points to the VSprite. ►

```
bob.BobVSprite = &vsprite;
```

To have the BOB automatically save and restore the background as it moves (so the entire background need not be redrawn each frame), set the SAVEBACK flag in the Flags field of the VSprite, and allocate a block of chip memory large enough to store the area covered by the BOB. The memory must be able to store as many planes as there are in the playfield, regardless of the number of planes in the BOB. This pointer gets stored in the BOB's SaveBuffer field.

```
struct VSprite vsprite;
struct Bob bob;
struct RastPort rastport;

vsprite.Flags |= SAVEBACK;

long size = (long)sizeof(UWORD) * vsprite.Width * vsprite.Height *
rastport.BitMap.Depth;

bob.SaveBuffer = (WORD *)AllocMem(size, MEMF_CHIP);
```

By default, the entire rectangle of image data is used to replace the playfield imagery. To draw the BOB imagery through a mask (for a cookie-cut effect, which is the most common technique), set the OVERLAY bit in the VSprite's Flags field and point the BOB's ImageShadow field to a one-plane shadow mask (most-commonly the logical OR of all of the planes of the image) which is the same size as the BOB. If collision detection is being used, this can point to the VSprite's CollMask (or vice-versa). This data must reside in chip memory.

```
vsprite.Flags |= OVERLAY;

bob.ImageShadow = (WORD *)AllocMem((long)sizeof(UWORD) *
vsprite.Width * vsprite.Height, MEMF_CHIP);
```

or, if the VSprite's CollMask has been initialize

```
bob.ImageShadow = bob.BobVSprite->CollMask;
```

If you do not want the system to bother with saving and restoring the background imagery, set the SAVEBOB flag in the BOB's Flags field. If the BOB is part of an AnimComp, set the BOBISCOMP flag. Do not confuse the VSprite's Flags field and the BOB's Flags field. VSPRITE, SAVEBACK, and OVERLAY are VSprite flags. SAVEBOB and BOBISCOMP are BOB flags.

Several options are available for selecting colors for BOBs, making possible nifty special effects through a clever manipulation of the VSprite PlaneOnOff and PlanePick fields.

Typically, the Depth field provides all the information necessary. If your display and BOB are both five planes deep, and your desired color palettes for both match, you've got nothing to worry about. Your BOB can have fewer planes of data than the playfield, however, and by default, only the number of planes provided are copied into the display.

You can specify selectively which display planes your BOB imagery is copied into by setting bits in the PlanePick field (this is typically 0xFF, enabling all planes). If you have a two-plane BOB and a five-plane display, and you want plane zero of the BOB in plane zero of the display and plane one of the BOB in plane two of the display, set the first (bit 0) and third (bit 2) bits in PlanePick.

```
bob.Depth = 2;
bob.PlanePick = 0x05;  /* bit 0 | bit 2 -> 00000101 */
```

What do you do with the unused planes in the background? This is handled by the PlaneOnOff field. If there is a bit set in this field, 1s are copied into the plane specified by this bit. If this bit is clear, 0s are copied. If the OVERLAY bit is set and the ImageShadow mask is used in drawing the BOB, bits in the destination plane are set or cleared where the mask exists. If the OVERLAY bit is clear, the entire rectangle is set or cleared for that plane.

As an overview, image data is blitted into planes that have a bit set in the PlanePick field, 1s or 0s are set in planes that have bits set or cleared in the PlaneOnOff field.

If collision masks have been allocated for the VSprite, and the BOB's ImageShadow points to the VSprite's CollMask field, the system can automatically generate the image shadow mask for you with a call to InitMasks().

```
vsprite.CollMask = (UWORD *)AllocMem((long)sizeof(UWORD) *
                    vsprite.Width * vsprite.Height, MEMF_CHIP);
bob.ImageShadow = vsprite.CollMask;
InitMasks(&vsprite);
```

DOUBLE-BUFFERING

Double-buffering, by enabling essentially two drawing areas, is used to minimize flickering caused when graphics are being drawn into the display. While one is being displayed, the other is being drawn into, then they swap and the process continues. This way, you don't see complex objects being rendered, just the results, and you are greeted with smoother animation.

If BOBs are to be used in a double-buffered display, a DBufPacket extension structure must be initialized and assigned to the BOB's DBuffer field.

```
struct DBufPacket {
   WORD BufY, BufX;
   struct VSprite *BufPath;
   WORD *BufBuffer;
};
```

The BufBuffer field of the DBufPacket points to an area of chip memory large enough to hold the region of background imagery covered by the BOB (the same size as the BOB's SaveBuffer area).

```
struct Bob bob;
struct DBufPacket dbufpacket = {0};
long size;

bob.DBuffer = &dbufpacket;

size = (long)sizeof(UWORD) * bob.BobVSprite->Width *
bob.BobVSprite->Height * bob.BobVSprite->Depth;
dbufpacket.BufBuffer = (WORD *)AllocMem(size, MEMF_CHIP);
```

USING BOBS

Unlike VSprites, the drawing order of BOBs can be specified by chaining BOB Before and After pointers. To draw bob1 after bob2, bob1's Before pointer points to bob2, and bob2's After pointer points to bob1. If this seems confusing, it is! But due to historical reasons it's the way you have to do it.

Here it is again:

```
struct Bob bob1;
struct Bob bob2;

/* link the BOBs so bob1 is drawn before bob2 */
/* incidentally, bob2 is drawn after bob1 */
bob1.Before = &bob2;
bob1.After = NULL;
bob2.Before = NULL;
bob2.After = &bob1;
```

To add a BOB to the current GEL list, use the AddBob() function:

```
struct Bob bob;
struct RastPort rastport;

AddBob(&bob, &rastport);
```

When finished with the BOB, remove it with RemBob():

```
RemBob(&bob);
```

Actually, RemBob() is a macro that sets the BOBSAWAY flag in the BOB, and the BOB is removed by the system during the next call to DrawGList(). To remove the BOB immediately and unlink it from the GEL list, use RemIBob().

```
RemIBob(&bob, &rastport, &viewport);
```

You must call DrawGList() again after using RemIBob() to refresh any BOBs that were erased when the BOB was removed. As with VSprites, SortGList() and DrawGList() must be called to display the BOBs, and after changes to the BOB position or imagery. If you are managing the display yourself, but are not dealing with any true VSprites, just BOBs, you can safely omit the calls to MrgCop() and LoadView(), because no special Copper instructions need be generated for BOBs.

Between calls to SortGList() and DrawGList(), you can change dynamically the position of the BOB by modifying the X and Y fields of the BOB's VSprite structure; the imagery by changing the ImageData pointer (if you're using the BOB's ImageShadow, be sure to call InitMasks() on the BOB to update the mask); and the drawing priorities by modifying the Before and After pointers. The PlanePick and PlaneOnOff fields can also be changed on the fly to modify the BOB's colors.

COLLISION HANDLING

The GELs system offers two types of collision detection: GEL-to-GEL and GEL-to-boundary. To enable collision handling, you must allocate the collision-handler table and assign it to the GelsInfo CollHandler field. This table is really an array of 16 pointers to collision-handler functions. One handler is used for GEL-to-boundary hits, and the other 15 are used for GEL-to-GEL hits. The actual function called is based on the colliding GELs' HitMask and MeMask fields in the base VSprite structure. If a set bit in the HitMask of one GEL matches a set bit in the MeMask of the other GEL, the collision routine corresponding to that bit number is called.

If you wish your VSprite or BOB to automatically perform collision-handling functions when it runs into something, initialize it's base VSprite's collision handling fields:

```
vsprite.HitMask
vsprite.MeMask
vsprite.BorderLine
vsprite.CollMask
```

Note that these fields are all set to 0 to completely disable collision handling for that GEL. Two areas of memory must be allocated for each GEL for proper collision detection. CollMask points to a single plane of chip memory the size of the image.

```
vsprite.CollMask = (UWORD *)AllocMem((long)sizeof(UWORD) *
vsprite.Width * vsprite.Height, MEMF_CHIP);
```

BorderLine points to a single line of chip memory as wide as the image.

```
vsprite.BorderLine = (UWORD *)AllocMem((long)sizeof(UWORD) *
vsprite.Width, MEMF_CHIP);
```

These areas must reside in chip memory. CollMask will contain the shadow mask (a single plane with bits set where nonbackground pixels exist in the image), while BorderLine contains a "squashed-down" single-line version of the shadow mask for quick collision detection with boundaries.

You set collision-handling functions with the SetCollision() function.

```
ULONG num;
void (*routine)();
struct GelsInfo gelsinfo;

SetCollision(num, routine, &gelsinfo);
```

Here, num is the collision-handler number to assign routine to. This is the routine called when the respective bits are set in the HitMask and MeMask fields of the VSprite. gelsinfo is the pointer to the GelsInfo structure used to manage your GEL list.

A GEL-to-GEL collision-handler routine takes two pointers to VSprite structures as its parameters. The first is the upper-left GEL involved in the hit, and the second is the lower-right GEL. The upper-most GEL to the left is always the first GEL sent to the handler.

There can only be one GEL-to-boundary function, and this is collision handler number zero. This routine takes a pointer to the VSprite that hit a boundary, and a flag WORD with bits set indicating which boundaries were hit or exceeded. These bits are TOPHIT, LEFTHIT, RIGHTHIT, and BOTTOMHIT.

The boundaries for GEL-to-boundary collision detection are set in the GelsInfo structure.

```
struct GelsInfo gelsinfo;

gelsinfo.topmost = 0;
gelsinfo.bottommost = 200;
gelsinfo.leftmost = 0;
gelsinfo.rightmost = 320;
```

To actually test for GEL collisions, call DoCollision(). This is usually done immediately after the GELs are displayed, and will automatically call your collision handlers as needed (note that more than one may be called if more than one hit takes place, but only one handler will be called for any particular hit between two GELs).

See the VSprite program on the disk for a complete example of the collision-handling system.

GEL USER EXTENSIONS

You can attach your own data to VSprites, BOBs, and Anim- ►

Comps, allowing you to perform special processing as needed on your GELs. One common extension is acceleration and velocity support for VSprites and BOBs.

To do this, you define a structure which will contain all the extra information you require. Here is an example:

```
struct GELinfo {
    WORD x_vel;
    WORD y_vel;
    WORD x_acc;
    WORD y_acc;
};
```

To extend the VSprite structure with this information, perform the following define:

```
#define VUserStuff struct GELinfo
```

and to extend a BOB:

```
#define BUserStuff struct GELinfo
```

and to extend an AnimOb:

```
#define AUserStuff struct GELinfo
```

Note that you can not use the "C" typedef operator. You must use the define construct. The structure and the above macro must be defined before the gels.h header is included. The preprocessor will now actually include your structure as part of the respective GEL structure. Access to this structure is available through the GEL's UserExt field. If you do not provide an extension structure, UserExt is a WORD value and can be used for user information of that size if desired.

See the VSprite program on the disk for a complete example of user-extension handling.

"The Amiga provides four levels of frame animation: simple, sequenced, ring, and hierarchical motion."

THE ANIMATION SYSTEM

Finally, with the voluminous preliminaries out of the way, we will get into actual animation. As you are probably aware, animation is achieved by flipping through successive frames of images, each slightly different. If the frames are drawn properly, the result is fluid motion.

The Amiga provides four levels of frame animation: simple, sequenced, ring, and hierarchical motion.

Simple Motion is achieved simply by moving an image across the screen a little at a time. Missiles and falling safes are examples of animation achieved with simple motion.

Sequenced Motion is similar to simple motion, except as the object moves, its image changes as well. A rolling spoked wheel or a tumbling mahogany desk are examples of sequenced motion.

Ring Motion is similar to sequenced motion, except the object is usually anchored to something. Each image in a cycle of movement is drawn with respect to a common anchor point, and at the end of the cycle, the anchor point is updated based on a certain value. Imagine using sequenced motion to move a spoked wheel. What will happen if the velocity doesn't match the rate at which the frames are drawn? If the frames don't match up, and the object's translation value is not just right, the wheel will appear to slide across the surface upon which it is supposed to be only rolling. Ring motion can be used to draw all of the frames for one revolution of the wheel in a fixed space. In each frame, the wheel will be drawn shifted a little as well, corresponding to how far it has turned. When being animated, the wheel "object" remains in place for a complete cycle of its frames, although because of the way the frames are drawn, the it still appears to move. When the wheel has made one complete revolution, the wheel object's position is updated by a ring translation value, which corresponds to how far the wheel has actually travelled due to its turning.

Hierarchical Motion is achieved by complex linkage of sequenced and ring-motion frame sequences into one object. As the object moves, the individual animation sequences are also updated, and move with respect to the host object. For example, imagine animating a walking person. The head, arms, legs, and torso can each be sequenced motion frames, so as the person object is moved, the individual pieces move with the body and also swing back and forth and up and down as they normally would when a person walks.

The primary component of an animated object is the AnimComp:

```
struct AnimComp {
    WORD Flags;
    WORD Timer;
    WORD TimeSet;
    struct AnimComp *NextComp;
    struct AnimComp *PrevComp;
    struct AnimComp *NextSeq;
    struct AnimComp *PrevSeq;
    WORD (*AnimCRoutine)();
    WORD XTrans;
    WORD YTrans;
    struct AnimOb *HeadOb;
    struct Bob *AnimBob;
};
```

Each AnimComp is linked to a BOB, which in turn provides the imagery for that component. For any animation object, at least one AnimComp is used. AnimComps also contain offsets, XTrans and YTrans, that are added to the host animation object's location to determine the component's location. These are deltas, and can be positive or negative (or zero, for that matter). AnimCRoutine can be a pointer to a user function that is called during the animation stage. This function takes one parameter, a pointer to the AnimComp being handled. TimeSet is a counter indicating how many frames this component must be displayed before cycling to the next frame of the sequence.

Each animation object is represented by an AnimOb:

```
struct AnimOb {
    struct AnimOb *NextOb;
    struct AnimOb *PrevOb;
    LONG Clock;
    WORD AnOldY;
    WORD AnOldX;
    WORD AnY;
    WORD AnX;
    WORD YVel;
    WORD XVel;
    WORD YAccel;
    WORD XAccel;
    WORD RingYTrans;
    WORD RingXTrans;
    WORD (*AnimORoutine)();
    struct AnimComp *HeadComp;
    AUserStuff AUserExt;
};
```

The AnimOb contains the location of the object and a pointer to the head animation component. It also contains linkages for the master object list, a pointer to a function that is called when the object is calculated (optionally provided by the programmer, this routine takes a pointer to the AnimOb as its only parameter), ring translation values if the object uses ring-motion, and internal variables. Note also that AnimObs contain information for velocity and acceleration, but they still have user extension support.

There's one difference between the coordinate system used for AnimObs and AnimComps. All coordinates are referenced as 16-bit, fixed-point binary fractions with the decimal at position six. This gives positional and incremental precision down to $^1/_{64}$ of a unit. A constant, ANFRACSIZE, is provided for the proper shift value to get screen coordinates into this fraction format. Just shift all of your screen coordinate values to the left by ANFRACSIZE. If your AnimOb or AnimComp is being referenced from within a user function called by the Animate() function, don't forget to shift your object values back to the right by ANFRACSIZE to get the actual pixel location.

Animation sequences and objects are set up in several different ways, depending on what type of animation you wish to perform. To set up a simple-motion animated object, all you need to do is initialize one AnimComp and point its AnimBob pointer to the BOB you wish to use for the imagery:

```
struct Bob bob;
struct AnimComp acomp = {0};
struct AnimOb animob = {0};

/* initialize your BOB first */
bob.BobComp = &acomp;  /* link the BOB to its host
AnimComp */

acomp.AnimBob = &bob;
acomp.HeadOb = &animob;
acomp.XTrans = 0;
acomp.YTrans = 0;
/* this component will be drawn wherever the AnimOb
is drawn, since the offsets are zero */

animob.HeadComp = &acomp;
animob.AnX = 20  << ANFRACSIZE;  /* don't forget! */
animob.AnY = 150  << ANFRACSIZE;
/* the AnimOb is initially drawn at 20, 150 */
```

Now, when you move the AnimOb around, the specified BOB is drawn with it. This is essentially the same as handling the BOB yourself as a simple GEL, but the AnimOb handles things such as velocity and acceleration for you automatically.

To accomplish sequenced and ring-motion animation, you'll need several BOBs and animation components, all linked together with the AnimComp PrevSeq and NextSeq fields. PrevSeq points to the previous AnimComp in the sequence, and NextSeq points to the next one in the sequence. To loop the sequence, the last AnimComp's NextSeq points to the sequence's first AnimComp, and the first AnimComp's PrevSeq points to the last AnimComp in the sequence. Note that because there are multiple AnimComps, each can have different XTrans and YTrans values, and therefore different drawing positions relative to the host AnimOb.

The only difference between the setup for sequenced and ring-motion animation is in how the frames are drawn, and specifying the RINGTRIGGER bit in the appropriate AnimComp Flags field. When that AnimComp is drawn, the AnimOb's RingXTrans and RingYTrans values are added to the AnimOb's AnX and AnY location values.

In both cases, the AnimOb HeadComp field points to the first AnimComp to be drawn in the sequence.

```
struct Bob bob1, bob2, bob3;
struct AnimComp comp1, comp2, comp3;
struct AnimOb animob;

/* a three-frame sequenced ring-motion animation */
bob1.BobComp = &comp1;
comp1.AnimBob = &bob1;
comp1.PrevSeq = &comp3; /* previous is the last frame */
comp1.NextSeq = &comp2;

bob2.BobComp = &comp2;
comp2.AnimBob = &bob2;
comp2.PrevSeq = &comp1;
comp2.NextSeq = &comp3;

bob3.BobComp = &comp3;
comp3.AnimBob = &bob3;
comp3.PrevSeq = &comp2;
comp3.NextSeq = &comp1; /* loop back to frame 1 */
comp3.Flags = RINGTRIGGER;  /* frame 3 moves the AnimOb */

animob.HeadComp = &comp1;
animob.AnY = 150  << ANFRACSIZE;
animob.AnX = 20  << ANFRACSIZE;
animob.RingYTrans = 0;
animob.RingXTrans = 20  << ANFRACSIZE;

/* the AnimOb internal location is moved 20 pixels to the
right when frame 3 is drawn. The next time around, the
object is drawn in the new location. */
```

Finally, for a complex hierarchical object, there can be more than one animation sequence in the object. In this case, multiple sequences of AnimComps are linked together with the NextComp and PrevComp fields. The first AnimComp in a sequence is linked into the other sequences with these fields, just as sequence frames are.

Another important consideration now is the drawing order of the respective BOBs that make up the frames. If not given an explicit drawing order, the system will draw them in sorted order as it comes to them. You must use the BOB's Before and After pointers to order your AnimComp drawing sequence. These pointers need only be assigned for the first AnimComp's BOB for each sequence, as are the NextComp and PrevComp fields. The first sequence's first AnimComp PrevComp field should be NULL, while the last sequence's first AnimComp NextComp field should also be set to NULL, indicating that there are no more sequences for this AnimOb. ►

The code below describes an AnimOb with three parts. The first is just one image and doesn't change, while the second part has two frames and appears behind the first part. The third has three frames and appears in front of both of the other parts.

```
struct Bob s1bob;
struct Bob s2bob1, s2bob2;
struct Bob s3bob1, s3bob2, s3bob3;
struct AnimComp a1comp;
struct AnimComp a2comp1, a2comp2;
struct AnimComp a3comp1, a3comp2, a3comp3;
struct AnimOb animob;

/* the Bob/AnimComp linkages are removed here for brevity.
Typically, these objects are allocated dynamically anyway */

a1comp.PrevComp = NULL; /* this is the first sequence */
a1comp.NextComp = &a2comp1;

a2comp1.PrevComp = &a1comp; /* second sequence */
a2comp1.NextComp = &a3comp;
a2comp1.PrevSeq = &a2comp2; /* next frame is frame 2 */
a2comp1.NextSeq = &a2comp2;

/* note that there is no PrevComp or NextComp linkage here.
We only use the FIRST AnimComp in the sequence for linking
multiple sequences */
a2comp2.PrevSeq = &a2comp1;
a2comp2.NextSeq = &a2comp1;

a3comp1.PrevComp = &a2comp1;
a3comp1.NextComp = NULL;    /* this is the last sequence */
a3comp1.PrevSeq = &a3comp3;
a3comp1.NextSeq = &a3comp2;

a3comp2.PrevSeq = &a3comp1;
a3comp2.NextSeq = &a3comp3;

a3comp3.PrevSeq = &a3comp2;
a3comp3.NextSeq = &a3comp1;

animob.HeadComp = &a1comp;
/* now specify the sequence frame drawing order */
/* sequence 2 (two-frames) first */
a2comp1.AnimBob->After = NULL;
a2comp1.AnimBob->Before = &a1comp.AnimBob;

/* now draw sequence 1 (the single-frame part) */
a1comp.AnimBob->After = &a2comp1.AnimBob;
a1comp.AnimBob->Before = &a3comp1.AnimBob;

/* and finally sequence 3 (three frames) */
a3comp1.AnimBob->After = &a1comp.AnimBob;
a3comp1.AnimBob->Before = NULL;
```

Note that it would be possible to have more than one component in more than one sequence of a ring-motion trigger frame. And be careful that you only specify one AnimComp as the ring trigger frame.

The system keeps track of all of the animation objects in a list using an AnimKey pointer, which is simply a pointer to the most recently added AnimOb. You initialize this pointer as follows:

```
struct AnimOb *animkey;

InitAnimate(&animkey);
```

Once your object has been set up properly, you can add it to the GEL system object list with AddAnimOb().

```
struct AnimOb animob;
struct AnimOb *animkey;
struct RastPort rastport;

AddAnimOb(&animob, &animkey, &rastport);
```

You call the Animate() function to update all of the object positions and sequence frames before calling the other normal GEL drawing routines. Animate() also calls any AnimOb or AnimComp user functions that may have been attached to the AnimCRoutine and AnimORoutine fields.

```
Animate(animkey, &rastport);
SortGList(&rastport);
DoCollision(&rastport); /* if you deal with collisions */

/* your collision handling functions may have reordered GEL
priorities, so be sure to call SortGList() again so the
system can put them in the right drawing order */

SortGList(&rastport);
DrawGList(&viewport, &rastport);‹f‹
```

VELOCITY AND ACCELERATION

Velocity and acceleration are handled automatically for AnimObs by the system. The XVel, YVel, XAccel, and YAccel fields are all referenced as fixed-point binary fractions.

For each call to Animate(), XAccel is added to XVel and YAccel is added to YVel, then XVel is added to AnX and YVel is added to AnY, thus changing the AnimOb's location. For example, to move an AnimOb right one pixel for each call to Animate(), with acceleration of one pixel every 64 calls, you would set up the values as shown:

```
animob.XVel = (1<<ANFRACSIZE);  /* one */
animob.XAccel = 0x0001;     /* 1/64th */
```

See the program Demo for a complete example of simple and hierarchical animation, double-buffering, and user Copper-list techniques.

YOUR NEXT MOVES

As you can see, a lot of effort was put into the Amiga's animation system. Although it is quite powerful, it's not really that hard to program when you look at it all from a completely object-oriented point of view. The example programs should give you plenty of information about specific types of GELs and animation support, and if that isn't enough, the *Amiga ROM Kernal Reference Manual: Libraries* (Addison-Wesley) and the Amiga includes and autodocs are invaluable reference sources. ■

Paul Miller has been an Amiga developer since 1985. Recently he became involved in developing CDTV applications, as well. In his spare time, he uses his Amiga for graphic design and music composition. Write to him c/o The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458, or contact him via Internet (pmiller@csugrad.cs.vt.edu).*

ON DISK

ARexx is for Writing Applications: Part II

"Just a reminder" to finish the calendar utility you started programming in the last issue.

By Marvin Weinstein

AREXX IS WELL SUITED to writing complicated applications, or so I contended in part one of this article (p. 26, January/February '92). As evidence of the fact, I offered a calendar/reminder utility written in ARexx, outlined the application, and detailed the calendar program. In this article, I provide final proof by describing the various programs, commands, and routines that comprise the reminder portion of the utility.

In addition to calendar.rexx, covered in part one, the calendar/reminder utility consists of an interface program and a number of housekeeping programs. The longest and most complicated is setremind.rexx, which provides an Intuition interface for creating and saving reminders. The four remaining programs handle housekeeping details, and should typically be included in the user's startup-sequence: Doreminders.rexx launches current-day reminders; cleanreminders.rexx erases expired reminders; for users who don't reboot their Amiga on a daily basis, reminddaemon.rexx runs the launch and clean programs once a day; and finally, remind.rexx posts the reminders at their scheduled times.

INSTALLING AND RUNNING THE REMINDER UTILITY

To install the reminder utility, decompress the file in the Weinstein drawer on the accompanying disk. Now, from the rexx directory copy the files setremind.rexx, remind.rexx, doreminders.rexx, cleanreminders.rexx and reminddaemon.rexx to your rexx: directory. Finally, copy all of the files from the libs directory to your libs: directory.

An environment variable, reminderdirectory, specifies where reminders are kept. Although you could set this variable by hand, setremind.rexx will handle this chore for you. (Please note that if you are not running AmigaDOS 2.0, you must assign the logical device env: to some directory. If setremind.rexx fails to find an assignment for env:, it will not run.)

To get started, go to a CLI and type:

```
rx setremind
```

The program first checks to see if reminderdirectory has already been defined. If not, the program leads you through the process of defining the variable and creating a directory in which to store future reminders. Once the variable is defined, the custom requester will appear (see Figure 1).

With the set-up complete, creating a reminder is simple. In addition to scheduling reminders for a specific day and time, you can have them recur at the same time every day or on a specific day each week. To schedule a daily reminder, type the word "everyday" in the string gadget that appears to the right of the Date: button. To schedule a reminder that recurs at a specified time on a given day each week, simply type the weekday you want: Monday, Tuesday, and so on. Only the first two letters of these words count and case is unimportant.

To erase expired reminders go to a CLI and type:

```
rx cleanreminders
```

To launch all reminders scheduled for the current day go to a CLI and type:

```
rx doreminders
```

You could add these commands to your startup-sequence to guarantee the operations occur each time you boot. If you don't boot your Amiga on daily, then instead add the line:

```
run rx reminddaemon
```

to your startup-sequence. This program runs cleanreminders and doreminders when the system clock strikes midnight.

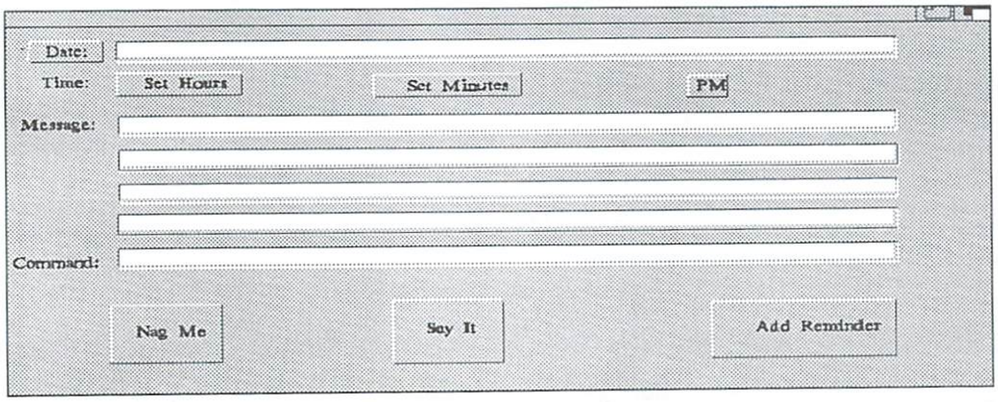


Figure 1: The reminder program's custom requester.

Once running, it is difficult to kill reminddaemon. To do so, go to a CLI and type:

```
setenv remindquit 1
```

Now, use AmigaDOS' Status function to find the processes that are loaded as wait, and send each one a break. One will be the reminddaemon. For example, if you type:

```
status
```

and find a line that says something like:

```
Process 6: Loaded as command: wait
```

you should type:

```
break 6
```

Of course, the process number need not be 6. In the event that several reminders are scheduled, they will also be listed as command wait. Sending a break to a waiting reminder forces it to post its message immediately. If remindquit is set to 1, then sending a break to the reminddaemon will post a message saying that it is shutting down.

SETREMIND'S MAIN PROGRAM

This is a skeleton outline of the program setremind.rexx:

```
/** rexx:setremind.rexx
*
**/
  call addlib('rexxarplib.library',0,-30,0)
  if showlist('p',REMINDERHOST) then exit 0

FRESHSTART:

  reminderdirectory = CheckSetUp()
  call GetDateTime()
  call MenuWindow(REMINDERHOST,REMINDERPORT)

RESTART:
        .
        .
  do forever
    if quitflag = 1 then leave
    t = waitpkt(REMINDERPORT)
    do ff = 1
      p = getpkt(REMINDERPORT)
      if c2d(p) = 0 then leave ff
      command = getarg(p,0)
      parse var command command sc1 sc2 sc3 .
      if command = "CLOSEWINDOW" ... then
        text = getarg(p,1)
      t = reply(p, 0)
      select
        when command = CLOSEWINDOW then do
          .
        end
        when command = "GADGETUP"
        & sc1 = "ADD" then do
          .
        end
          .
        when command = "GADGETDOWN" ,
        & sc1 = "SETHRS" then do
          .
        end
          .
          when sc1 = "DATE" then
          call ActivateGadget(REMI.. ,"MES1")
          .
          .
          .
        when command = "GADGETUP" ,
        & sc1 = "CALENDAR" then
          address AREXX calendar REMINDERPORT
        when command = "DATEIS" then do
          .
        end
        otherwise nop
      end
    end
  end
exit
```

The program begins with the obligatory comment and the, by now, familiar line that adds rexxarplib.library to ARexx's internal list. The next line of code uses ARexx's versatile showlist() function to check for a RexxArpLib host called REMINDERHOST, because you don't want to open the reminder-creation interface more than once. Because showlist() is not properly documented in the ARexx manual or in the original 2.0 documentation, it is worth discussing here.

AN ASIDE ON SHOWLIST()

Showlist() returns information about shared system lists. The version provided with OS 2.0 (ARexx 1.15) has the syntax:

```
Usage: showlist( option, [name], [separator])
```

where the options are A, D, H, I, L, M, P, R, S, T, V, W, which respectively stand for: Assigned directories, Devices, Handlers, Interrupts, Libraries, Memory-list items, Ports, Resources, Semaphores, ready Tasks, Volume names, and Waiting tasks. These options are useful when writing installation programs in ARexx.

Separator, the third argument (optional), specifies a character to place between the entries in the string showlist() returns. For example, to produce a list of libraries wherein the names are separated by a % symbol, type:

```
say showlist('l',,'%')
```

(Both commas separating the first and third arguments are required.) You can even use the hex character "0a"x as the separator, in which case a linefeed will be inserted between each entry on the list. Thus:

```
say showlist("v",,"0a"x)
```

produces a list of the form:

```
RAM_0
RAM DISK
WORK
AWTECH
```

These options are particularly useful when one of the listed items contains a space, as in RAM DISK above.

THE MAIN PROGRAM, CONTINUED

Next, there are two label clauses, FRESHSTART: and RESTART:. These clauses define where ARexx will resume execution after encountering the instructions

```
SIGNAL FRESHSTART
SIGNAL RESTART
```

found in the subroutines CheckSetup() and ProcessMessage().

You will notice new material in these subroutines and in the subroutines GetDateTime() and MenuWindow(). I will discuss the new material later in the article. For now, you only need to know that when MenuWindow() returns successfully, it will have created REMINDERHOST (a RexxArpLib host) and opened a message port named REMINDERPORT.

The structure of the main loops is similar to that used in earlier examples, with some significant differences. Therefore, let's review the basic procedure for handling messages generated by an RexxArpLib host and compare it to the procedure for handling messages generated by the Set Hours and Set Minutes gadgets.

Message handling occurs within the main do forever loop. An if..then statement at the top of this loop checks the variable quitflag. If quitflag = 1, the program leaves the loop, thereby ending the program. If quitflag ≠ 1, the next statement executed is a call to ARexx's waitpkt() function. This causes the program to sleep until a message arrives at REMINDERPORT. When that happens, the program wakes up and executes the statements contained within the do ff = 1 loop. Using a *do name = 1* construction lets you refer to the loop by name in a *leave name* instruction. This helps avoid confusion, particularly because the leave instruction is used in more than one place.

It is important to note that the inner loop, do ff = 1, keeps pulling messages from REMINDERPORT until no messages are left. Remember, after getpkt() is called, any messages queued at the message port will be ignored by subsequent calls to waitpkt(). In fact, calling waitpkt() before emptying the queue will seriously delay the handling of input events. (See"Custom Interfaces With ARexx," p. 28, November/December '91.)

If a message is queued at the REMINDERPORT getpkt() returns a valid address. If no messages are queued, getpkt() returns the hex character "0000 0000"x. The statement:

```
if c2d(p) = 0
```

checks to see that getpkt() has found no waiting messages. The function c2d() converts the hex string to a decimal number. I use this construction because it works with all versions of ARexx. With newer versions of ARexx you can compare the value of p directly to the value returned by the ARexx function NULL(). When a non-NULL address is returned by getpkt(p), the function getarg(p, number 0-15) extracts the information stored in the waiting message. (Recall that a general ARexx message has 16 slots that can contain information and that getarg(p) is equivalent to getarg(p,0).)

All of the button and string gadgets associated with the custom requester are created in the MenuWindow() subroutine. If you examine this routine you will see that the button gadgets generate messages that contain information in only the 0 slot, whereas string gadgets generate messages that contain information in both the 0 and 1 slots. Don't forget that failing to reply to a message after extracting its contents is an invitation to disaster.

In general, the 0 slot of the ARexx message generated by clicking on a gadget contains a string consisting of up to four words. The first word will be either CLOSEWINDOW, the name assigned to the gadget when it was added to the requester, GADGETUP, GADGETDOWN, or MOUSEBUTTONS. After the first call to getarg(p) the variable command will contain this string. I use ARexx's parse var instruction:

```
parse var command command sc1 sc2 sc3 .
```

to separate this command into its component parts and at the same time assign the first word to command, the second to sc1, the third to sc2, and so on. The dot that appears at the end of the line is a placeholder symbol indicating that any words that remain should be thrown away. It guarantees that sc3 will not contain a leading space. Placeholder symbols can also be used to discard words appearing at other places in the string to be parsed. Thus:

```
parse var command . sc1 sc2 sc3 .
```

would parse the string command and throw away the first word, assign the second to sc1, assign the third to sc2, the fourth to sc3, and then throw away any material that follows.

HANDLING NEW KINDS OF MESSAGES

Once the variable command is parsed into its component parts, control passes to the select.. when.. otherwise section of the program. While the procedure used in this section is similar to that used in previous examples, significant differences arise because our RexxArpLib host is now sending GADGETDOWN and MOUSEBUTTONS messages in addition to GADGETUP messages. These messages are needed because the Set Hours and Set Minutes gadgets now work (for the user) like the buttons on a digital clock: Placing the mouse pointer over the Set Hours gadget and pressing the left mouse button cycles the text through the numbers 1–12.

Setting the GADGETDOWN flag in MenuWindow() tells the host to send a message whenever the user places the pointer over a button gadget and presses the left mouse button. This is in addition to the GADGETUP message that is sent when the mouse button is released while the pointer is over the same gadget. But what if the user changes his mind and moves the pointer away from the gadget before releasing the mouse button? The MOUSEBUTTONS flag handles this event by indicating that a message should be generated whenever the user depresses or releases a mouse button when the pointer is not located over gadget. Because this is what the user normally does if he changes his mind, the program shouldn't respond to a message of this sort. Such a message is only necessary if the user starts the Set Hours (or Set Minutes) counter running and then moves the mouse pointer off the button before releasing it. In this case, if the program did not get a MOUSEBUTTONS message, the counter would keep running. The code that handles the other gadgets ignores MOUSEBUTTONS messages.

HANDLING THE CLOCK GADGETS

The following code shows how the Set Hours gadget is handled. It provides an example of using getpkt() to poll REMINDERPORT at high speed. Once the program enters the do icount = 1 loop, it simply pulls messages and ignores those that don't begin with GADGETUP or MOUSEBUTTONS. Once such a message is found, it immediately leaves the icount loop. It is safe to leave the loop without taking any further action, because control transfers back to the do ff = 1 loop, which keeps polling REMINDERPORT until all waiting messages have been handled.

```
when command = "GADGETDOWN" & sc1 = "SETHRS"
then do
  do icount = 1
    lasthr = (lasthr + 1)//12
    call Move(REMINDERHOST,180,42)
    call Text(REMINDERHOST,...)
    call Delay 5
    p = getpkt(REMINDERPORT)
```

►

```
    if c2d(p) = 0 then do
        command = getarg(p,0)
        parse var command command rest
        t = reply(p,0)
        if command = "GADGETUP" ,
          | command = "MOUSEBUTTONS" then
            leave icount
      end
    end
end
```

THE AM/PM GADGET

Another technique that appears in the main program relates to the AM/PM gadget. The label on this button gadget is meant to cycle between AM and PM whenever it is clicked. Because RexxArpLib doesn't provide a mechanism for simply updating the text of a button gadget, the programmer has to remove the old version of the gadget and then restore it after modifying the text. I have discussed this technique in conjunction with updating the contents of a string gadget. In this case, however, it is also necessary to erase the old gadget imagery before adding the new gadget. The sequence of steps in the following code shows how I accomplish this:

```
when command = "GADGETUP" & sc1 = "AMPM"
then do
   ampmcount = (ampmcount+1)//2
   if ampmcount = 0 then ampmtext = "AM"
   else ampmtext = "PM"
   call RemoveGadget(REMINDERHOST,AMPM)
   call SetAPen(REMINDERHOST,3)
   call RectFill(REMINDERHOST,460,35,476,44)
   call AddGadget(REMINDERHOST,....)
   call SetAPen(REMINDERHOST,1)
end
```

Notice the use of the RexxArpLib function RectFill() to erase the old gadget imagery. This is preceded by the call to SetAPen(), so the rectangle used to erase the imagery will be filled with the background color. Once the old gadget is erased, the RexxArpLib host is instructed to add back new gadget, and then to set the APen back to its original value.

CYCLING THROUGH THE STRING GADGETS

The final new technique used in the main program involves cycling through the string gadgets when the user hits a carriage return. Implementing this feature is simple, if you remember that hitting a carriage return while in an active string gadget results in a GADGETUP message. Because MenuWindow() defines string gadgets to return their name in the 0 slot, all you have to do is add such a line as:

```
when DATE then
   call ActivateGadget(REMINDERHOST,MES1)
```

This activates the gadget named MES1 whenever the user presses Return while the gadget named DATE is active. See the listing on disk for additional comments and refer to RexxArpLib.doc for the syntax of all RexxArpLib commands.

THE SUBROUTINES

The subroutines found after the main section of setremind.rexx introduce some new material, as well.

The subroutines used in previous articles were fairly simple, so I did not worry about subroutine variables having the same name as variables used in the main program. Once an ARexx program begins to get complicated, it is important to be able to limit the scope of a variable to a specific subroutine. This can be done by defining the subroutine as a PROCEDURE, which is accomplished by inserting REXX's PROCEDURE instruction after the label that indicates the beginning of the subroutine. For example:

```
WriteFile: PROCEDURE
        .
        .
return 0
```

The word PROCEDURE can appear anywhere within the body of the function, but you should put it on the same line as the label or on the next line. Declaring a subroutine as a procedure means that variables in the main program are protected from alteration by a call to the subroutine. Although keeping variables local to a subroutine makes for safer programs, programmers often wish to give the subroutine selective access to specific variables. The EXPOSE subkeyword provides REXX with a mechanism for doing this. For example:

```
WriteFile: PROCEDURE EXPOSE lsayit ... nag ...
```

says that the variables contained in the list following the word EXPOSE will be available to WriteFile() for use and modification. The listing provides examples of the use of the PROCEDURE instruction with and without the EXPOSE keyword.

A USEFUL TRICK

In complicated programs, defining a list of global variables at the beginning of a program, then having subroutines use the same list makes it easier to organize. A sneaky use of REXX's INTERPRET instruction makes this possible. For example:

```
        .
globals1 = "var1 var2 var3 var4"
        .
        .
WriteFile: interpret expose globals1
        .
```

makes var1, var2 and var3 available for use in WriteFile().

CHECKING THE USER'S SETUP

The subroutine CheckSetUp() looks to see if the environment variable reminderdirectory is defined. If the variable is not, CheckSetUp() leads the user through the process of defining this variable and creating a directory in which to store future reminders. Determining whether a file, directory, or logical device exists is easily done with the ARexx function exists(filename), as in:

```
say exists('env:reminderdirectory')
```

However, if the device referred to does not exist, or has not been assigned, a call to this function causes an automatic requester to appear. This is annoying if you want to check for a file without letting the user know. Fortunately, the latest version of the infamous Pragma() function provided with ARexx 1.15 lets you shut off the automatic requester.

Pragma() is a kind of grab bag function that handles an assortment of jobs related to extracting and changing the attributes of the system environment. I used this function in the first installment of this article to generate a unique Id for each invocation of calendar.rexx. In CheckSetUp() I use PRAGMA() to shut off automatic requesters by calling it with the arguments:

```
call Pragma('W','Null')
```

To turn automatic requesters back on, I issue the call:

```
call Pragma('W','WorkBench')
```

The remainder of CheckSetUp() uses techniques familiar from previous articles. Refer to the comments contained in the listing for details. Note in particular the use of RexxArpLib's getenv() and setenv() functions to check for and write environment variables.

THE SIGNAL INSTRUCTION

REXX's SIGNAL instruction can be used in two ways. First, it can control the state of the program's internal interrupt flags. In this case the SIGNAL instruction must be followed by the keyword ON or OFF, and a one of the condition keywords, such as BREAK_C, SYNTAX, ERROR, NOVALUE, and so on. This mode of operation allows you to take full control of what happens after the ARexx interpreter encounters an error, finds an undefined variable or one of the other specified conditions. Although this can be extremely useful, I have not employed it in the calendar/reminder facility.

In its second use, the SIGNAL instruction can be a kind of goto statement; it is used to transfer control to a label statement located elsewhere the same subroutine, or in the calling program. Thus, if there is a label clause:

```
FOOBAR:
```

located somewhere in an ARexx program, the instruction:

```
SIGNAL FOOBAR
```

causes execution to transfer to the label clause.

Be careful: The SIGNAL instruction is only *somewhat* like a goto statement. The actual purpose of SIGNAL is to permit flexible handling of error conditions; it is not meant to be a general-purpose programming tool. When ARexx encounters a SIGNAL instruction, it completely dismantles all active control statements, including IF, DO, SELECT, INTERPRET, and interactive TRACE, before transferring control to the label clause. Attempting to use SIGNAL FOOBAR to transfer control to another point within a do loop or select..when.. structure is doomed to failure.

Only the control structures belonging to an executing subroutine are dismantled when a SIGNAL instruction is encountered. This means that using SIGNAL to transfer control from a subroutine to a label found in a calling program is safe and that is why I can use SIGNAL as I do in the subroutines CheckSetUp(), ProcessMessage(), and BadDate(). (See the listing for additional comments.)

MAKEMENU

The MakeMenu() subroutine is much like its counterpart in calendar.rexx, with a few significant differences. One is that I ask for GADGETUP and MOUSEBUTTON events by adding these options to the line defining the variable idcmp that is passed to RexxArpLib's OpenWindow() function. For example:

```
idcmp = "CLOSEWINDOW+GADGETDOWN"
idcmp = idcmpll"+GADGETUP+MOUSEBUTTONS"
```

Another difference is that the calls to AddGadget() create message strings that are more complicated than in previous examples, such as:

```
AddGadget(.. ,"CALENDAR","Date:","%l %d")
AddGadget( .. ,"DATE", .. ,"%l %d%1%g",500)
AddGadget( .. ,"%d%1%g",500)
AddGadget( .. ,"%l %d%1%d")
```

In each of these cases I use the % mechanism provided by RexxArpLib to specify the structure of the message to be returned, see rexxarplib.doc for details. Briefly, %l translates to the GADGETUP, GADGETDOWN, or MOUSEBUTTONS depending upon the IDCMP class of the event; %d translates to the name of the gadget that caused the event; %1 says put the material to follow in the first slot of the message; %g, used only for string gadgets, says "put the contents of the gadget in this place." Thus, clicking on the Date: gadget produces the message:

```
GADGETDOWN CALENDAR
```

and releasing the mouse button while the mouse pointer is over the same gadget produces:

```
GADGETUP CALENDAR
```

It is worth examining the use of ARexx's right() function to format the text that is written next to the Set Hours and Set Minutes gadgets. This function, as well as left(), center(), copies(), overlay(), insert(), substr(), and subword(); are extremely useful when the program has to produce formatted text.

GETVAR AND WRITEFILE

The GetVar() subroutine, familiar from previous articles, is called in WriteFile() to read all string gadgets before writing the reminder. This procedure enables the program to pick up any changes the user might have made in a string gadget without hitting a carriage return.

When you look at the listing, note the use of ARexx's open(), close(), and writeln() functions to create the permanent copy of the new reminder. When using ARexx's open() function, programmers often fail to specify its third argument, either R or W, to indicate whether the file is to be opened in Read or Write mode. If this argument is not specified ARexx will, by default, open the file in Read mode and subsequent calls to writeln() will fail.

HOUSEKEEPING PROGRAMS

The programs doreminders.rexx, remind.rexx, cleanreminders.rexx, and reminddaemon.rexx are short ARexx programs designed to handle specific chores. The only two functions used in these programs that have not been discussed are explained in the ARexx manual, in the OS 2.0 documentation, and in the comments included in the listings. The most interesting aspect of these programs is the way they use ARexx's ability to launch AmigaDOS commands to achieve their ends. For example, remind.rexx speaks the reminder message by writing it to an external file and then sending it to SPEAK:.

Whether or not it helps you remember your mother's birthday or be on time for staff meetings, this calendar/reminder program is convincing proof that ARexx can be used for creating serious applications. ■

Marvin Weinstein uses ARexx and REXX extensively in his work at the Stanford Linear Accelerator. Contact him c/o The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458, or on BIX (mweinstein).*

Designing the User Interface: Text Fonts

By David "Talin" Joiner

ONE THING THAT always impresses me is an application's font awareness (or lack thereof). A number of programs look terrible on my system, simply because the programmer didn't consider that the application might be run on a system with a default font other than topaz-8.

The simplest way to avoid this problem is to hardcode a font into your application. With this method, however, the window's title-bar font is taken directly from its screen font, and you can't change the font for the Workbench screen. (Not on my Workbench you don't!) Therefore, you have to open a custom screen, which isolates your program from the other Workbench windows. I call it "The Custom Screen Leper Colony."

A more advanced method entails repositioning your on-screen elements (buttons, gadgets, and so on) based on the font size. It sounds like a lot of work, but is actually easier than the methods many of us are accustomed to.

For example, I used to design my windows and screens in DeluxePaint. Once the screen was finished, I would painstakingly write down the coordinates of every gadget, outline box, or static text, and then write the corresponding code into my program—a tedious task.

With the new method, when I want a vertical column of gadgets, I write a function that iterates through the gadget list, placing each gadget just below the last. Modifying the function to adjust for the font size is easy. The result: I never have to write down the coordinates—the program figures them out for me at runtime.

For more complex layouts you need to define a relationship between the various parts. For example, if I have a scrolling list next to a vertical column of buttons, my program loops through all of the buttons' text labels to see which is the largest, in pixels. After adding a reasonable amount of space to that for borders and inter-gadget spacing, it lays out the column of gadgets. This gives me the column height. Because the scrolling list is to the right of the column, I already know the left edge, top edge, and height of the scrolling list. All that remains to determine is its width, which can be either fixed or based on the width of the window (if that is known).

Sometimes I work from a fixed window size (as when I clone the Workbench screen size), and other times I use a variable window size, making it just big enough to hold all the gadgets. Note that under OS 2.0, screens can actually be larger than the video display. Because you probably don't want to open your window larger than the actual video display, you'll need to look at the DisplayClip variables that are accessed through the new ViewPortExtra functions in the graphics library.

Next, consider the window's title bar, which will change size based on the screen font. If the window is already open, determining the title bar size is easy: Just look at the size of the top border. Otherwise, you can use the screen's BarLayer size or add three to the screen's font size. I've been told that the constant 3 is not likely to change in the future.

You'll need to adjust your menus to the font as well. See "Menus for a New Generation" (p. 4, April/May '91) for ideas about accomplishing this.

TEXT SUPPORT

Fonts also affect the display of documents. In general, there are four possible levels of font support within a document:

1. No font support: Document font is a hardcoded, monospaced font. (old programs)
2. Monospace font support: The document font can be any one, monospaced font. (text editors)
3. Proportional font support: The document font can be any one font. (outline processor or low-end word processor)
4. Multifont support: The document may contain more than one font. (full-featured word processor or desktop publisher)

At levels 3 and 4, rendering speed becomes a problem. As we all know, when rendering, calculating word wrap, and clipping to document borders, proportional fonts are slower than monospaced fonts. Fortunately, there is a handy way to speed up rendering and simplify clipping. This technique works in any application that doesn't support multicolored fonts (it can work for multicolored rendering, but it's trickier).

Basically, you create an off-screen bitplane that is as tall as the largest font used in the program, and as wide as the line to be rendered (which can be wider than the window's drawing area). If you don't know what these dimensions will be, estimate the size and reallocate the bitplane later if needed, for a slight speed penalty.

Each time you render a line of text, use the normal Text() call into the off-screen bitplane. If your application supports more than one font size, you'll probably first want to clear the bitplane using the Blitter. For even greater speed, create your own text-rendering routines that write directly to the off-screen bitmap. (If your characters are small, it's faster to use the CPU than the Blitter to draw them).

Once the characters are rendered, you need to clear the buffer from the end of the last character to the end of the buffer. The call ClearEOL() in the graphics library will do it for you.

Make your programs sensitive to font choices, for the user's sake. Reposition your buttons, menu items, and gadgets based on the user's chosen font size.

Finally, use BltTemplate() to blast the bits from your off-screen buffer to your window's drawing area. You don't need to create a clip region for this, simply adjust the coordinates of the source and destination bitmaps/rastports.

If your text is always drawn in color 1, you can optimize even further by rendering only into the lowest bitplane of your window. (You'll need to insure that the other bitplanes are kept cleared, although you can use them to highlight text.) To do this, make a copy of the window's rastport and set that copy's rp_Mask to 1. Then use the copied rastport to do the BltTemplate(). The other bitplanes are never touched. (Another trick: If you change pens or other rastport fields a lot, you can save time by keeping several copies of the rastport around, each with its own settings.)

FONT MENU VS. FONT REQUESTER

Originally, most Amiga programs let the user select fonts from a menu. As users accumulated more fonts, however, the menus became so large that they ran off the bottom of the screen. Even worse, because menus are not clipped to the window boundaries for speed reasons, this sometimes caused system crashes and blanking displays.

Scrolling menus like those on other graphical user interfaces seem like a possible solution. The Amiga currently doesn't support scrolling menus, though, and there are a number of user interface experts that think scrolling menus are a bad idea.

The officially adopted solution is to have a font requester that allows the user to browse though the fonts using a scrolling list, much like a file requester. In fact, the new operating system provides a font requester as part of the ASL library. (For details on asl.library, see "Easy File and Font Requesters," p. 7.)

EMBEDDED FONTS

Sometimes you'll want a unique font for your application (for example, a font of musical symbols for a music program). You can avoid the clumsy step of copying the font to the user's font drawer by embedding it within your application. There are a number of freeware utilities that convert fonts to C source code or assembly language. You can then use the font directly in your application without opening it. You'll probably need to make sure, however, that the actual font data is in chip RAM. From what I've determined, some versions of the OS require the font data to be in chip RAM, while others don't.

One caveat: Under 2.0, when you SetFont() a font to a rastport, extra information for that font is allocated. Normally, this information is deallocated when you close a font, but because embedded fonts are never closed, you will lose about 20 bytes of memory. You can forcibly deallocate the extra information using the new 2.0 routine StripFont().

FONT FUTURE

There are a couple of advanced font features that may apply to your application. The first is ColorFonts, a standard for those fonts containing more than one color. The ColorFont standard is part of the OS 2.0 (1.3 had a ColorFont utility to take care of it). The standard graphics library routine Text() will automatically handle the details of rendering a color font, but there are some details, such as color remappings and palette selections, that your application might want to deal with in the context of a ColorFont.

Outline fonts, the second new feature, are fonts that are stored as polygons rather than as bitmaps. Commodore has been working on the technology to bring outline fonts to the Amiga, and direct support for them is available in OS 2.0. Outline fonts are accessed like normal fonts. (When you open an outline font, it creates a normal bitmap font in that point size. When you close it, the bitmap representation is deallocated.) However, outline fonts can be opened in any point size. Your font requester should be able to detect an outline font and allow the user to enter any size, rather than just the standard sizes listed.

WRAPPING UP

As with many aspects of 2.0, with fonts you can no longer count on constant defaults. Build flexibility into your programs and you never need visit the Custom Screen Leper Colony. ■

David "Talin" Joiner is the author of Music-X and Faery Tale Adventure, plus an artist, award-winning costume designer, and moderator of the user.interface topic of the Amiga.sw BIX conference. Contact them c/o The AmigaWorld Tech Journal, *80 Elm St. Peterborough, NH 03458, or on BIX (talin).*

ON DISK

Utilizing the Revision Control System

RCS can help save time, hassles, and money during program development.

By Bryce Nesbitt

MANY TOOLS CLAIM to increase productivity, but not all live up to their promise. In almost ten years of continuous use in large UNIX-based environments, the Revision Control System (RCS) has proven its value. Written by Walter F. Tichy and made available free of charge through his and Purdue University's kindness, RCS has been entrusted with the crown jewels of hundreds of major projects, including the source code for all versions of the Amiga operating system. Now the advantages of a large-systems development tool are available to Amiga programmers, thanks to Rick Schaeffer and Raymond S. Brand, who ported it over.

Don't be scared by the word "control." RCS is unlike some so-called CASE (Computer-Aided Software Engineering) tools that are merely thinly veiled attempts at Orwellian control. Unlike those tools, RCS does not force you to think, act, program, and operate in a certain manner. Designed for programmers by programmers, the defaults of RCS perform reasonable actions. If the defaults are inappropriate, you can override or subvert RCS operations. RCS won't get in your way, at least not for long.

WHAT'S IN IT FOR ME?

RCS acts as librarian for your source code, offering commands for check in, check out, and housekeeping activities. In addition, RCS handles all file management and tracking. Previous versions of source code are stored as "reverse deltas." This compact format keeps many past revisions of the source within convenient reach. The benefits may seem subtle at first, and it all may sound complex, but experience will quickly show the ease and advantages:

• RCS saves multiple revisions of your code seamlessly. Modifying a source file does not destroy the older revision. You can quickly recall or compare any arbitrary revision.
• RCS can mark an entire set of files as a "release." You can quickly revert to a release for comparison, regeneration, bug tracking, or any other purpose.
• RCS offers powerful branch and merge facilities (more on this later). Two parallel versions of your product can be under development simultaneously without the usual hassle or wasted work.
• RCS minimizes storage space. Rather than keeping several copies of your source code, RCS stores only deltas or differences. Unless you change every line of every file with each revision, the delta format is likely to be smaller.
• RCS provides access control for multiple programmers working on the same project, preventing two programmers from simultaneously overwriting each other's work. Members of a team can keep track of a project's progress by viewing RCS logs. With RCS, changes made by one programmer are instantly obvious to all others.
• RCS allows mistakes to be easily reversed. If a change turns out to be a disaster, simply ask RCS for the old version back. Even if you accidentally deleted a section, RCS can quickly recover the missing lines.
• RCS tracks the when and where of changes. Changes are logged into RCS with the time, date, and a short description. During development these log messages provide convenient reminders. Even years later these logs can be valuable to the people maintaining the original code.
• RCS allows you to create test or experimental versions without risk to the main-line development. For example, you might try a new approach, search for an empirical solution to a problem, or test out a new idea. Without RCS you could easily forget debugging information or "temporary" changes. Some compatibility problems with 2.04 Kickstart were caused by old code that nobody intended to leave in the product. RCS lets you check in intermediate versions, thereby protecting them from harm. When the test or experiment is complete the new source can be compared against the stable version, and desired changes automatically merged.
• RCS is a fantastic aid in debugging; you can test solutions to a problem without risk to the main source code. If a new problem has been introduced into a previously working project, revisions can be peeled away until the bug is located. Reviewing the source of problems can be instructive; was it late at night? Were there many distractions or phone calls?
• RCS for the Amiga is fully interoperable with RCS for other systems. RCS has a wide enough installed base to be a good system for distributing source code.
• RCS is a perfect companion to networks. The RCS files can be stored on a central server (available to all programmers), while each programmer maintains a local working area. RCS was, of course, designed from the start for multitasking multiuser computers.

INSTALLING RCS

At this point you should install RCS, which (along with code from Richard Stallman's Free Software Foundation) is found in the accompanying disk's Nesbitt drawer. Now, follow along with the example. Installing RCS is a snap. You'll need about 400K of hard-disk space and a few minor additions to your s:User-Startup file:

```
setenv USERNAME bryce
assign RCS: work:bin
```

```
RCS file:     RCS/hello.c,v;  Working file:   hello.c
head:         1.3
branch:
locks:        ; strict
access list:
symbolic names:
comment leader:  " * "
total revisions: 3;    selected revisions: 3
description:
My first C program.
----------------------------------------------------------------
revision 1.3
date: 91/11/10 15:01:23;  author: bryce;  state: Exp;  lines added/del: 2/2
Added loop counter printout.
----------------------------------------------------------------
revision 1.2
date: 91/11/10 14:59:54;  author: john;  state: Exp;  lines added/del: 4/1
Added loop to print string ten times.
----------------------------------------------------------------
revision 1.1
date: 91/11/10 14:58:36;  author: bryce;  state: Exp;
Initial revision
```

Figure 1. Sample output from Rlog.

USERNAME is a Shell environment variable. RCS will tag all operations with this name. If multiple people will be accessing files, be sure that all names are unique. RCS prevents differing user names from modifying files in an inconsistent manner.

The "assign RCS:" command must point to the storage location for the RCS executables. If you are short of hard-disk space and don't need merging capabilities, you may delete the files diff3, merge, rcsmerge, and ident. Be warned, however, that RCS is not recommended for floppy-based systems.

Most UNIX programs, RCS included, depend on the availability of large or infinite stack space. You'll need a stack of 20K or greater to stay out of trouble. Add the following to your s:shell-startup:

```
stack 20000
```

IS IT REALLY THIS EASY?

The three most used RCS commands are ci (Check In), co (Check out), and RCSDiff (show differences). An example should make things clear. Assume you just created a file called hello.c:

```
cd RAM:
makedir RCS ;storage for RCS files
ci hello.c
```

RCS prompts you for a short description of the file. In this case the description might be "My first C program." The working file hello.c will be deleted; don't panic! The file is safe in the RCS library. Before viewing or modifying a file, you must first check out a copy. Check out may either be unlocked, –u, or locked, –l. You can modify only locked files:

```
co –l hello.c
```

After your edits and testing are done, you would typically compare the current version with the last check in. This step provides verification that you made only intentional changes and a mental jog for describing your changes:

```
rcsdiff hello.c
```

All differences are shown on the screen. If you are satisfied with your work, complete the loop by checking in the new revision:

```
ci hello.c
```

RCS prompts you for a brief description of the changes. Try this yourself; the Nesbitt drawer on disk contains a sample hello.c file you can play with.

WHERE IS IT HIDING MY CODE?

RCS processes RCS delta (or ,v) files. To save clutter, it is best to create a directory called RCS in each location you use RCS. By default, RCS appends ,v to your filename and stores the result in the RCS/ directory. The RCS delta file is the central repository for all information about a file. If you wish to view a file, you must request a "working file." Working files may be checked out unlocked or, for modification, locked. You perform all normal edit and view actions on the working file using standard text tools.

Upon completion of a release or major feature, you should check in the code. RCS compares the working copy with the previously stored version, and stores the differences. RCS can recreate any old version on demand.

The RCS delta file is stored in plain text. This can be comforting; even if some bug or problem were to arise with RCS, your code could easily be recovered by editing the delta file. The grammar of the delta file is fully described in the documentation in the Nesbitt drawer of the accompanying disk.

WHAT CAN I DO?

Ten binary commands and one script are provided with the RCS distribution. I'll describe each only briefly; for detailed documentation, consult the .doc files in the Nesbitt drawer. As you read about each command, give it a try on your Amiga. I included several sample RCS files on the disk.

rlog (RCS Log): Displays status, current revisions, lockers, branches, and so on. See Figure 1 for an example. The scope of the information printed by rlog can be filtered by date, revision numbers, user name, outstanding locks, or other criteria.

RCSDiff (RCS differences): Compares versions of a file. By default RCSDiff compares the current working file with the most recently checked-in version. RCSDiff can also compare against an older revision or compare two older revisions.

rcs: The master control program. Rcs has command line options for forcing locks, setting symbolic revisions, changing access lists, appending text, and so on. Unless your needs are obscure, you will use the rcs command infrequently.

ci (Check In): Deposits new revisions into the RCS ,v file. Ci automatically compares the files and complains if no changes have been made, if the file is locked by someone else, or if the file was never locked at all. By default the revision number is bumped; if the previous revision was 1.0, the next will be 1.1. You can force a new revision number with the –r option. By default, ci deletes the working file. The –l or –u options cause ci to preserve a locked or unlocked copy. For convenience, I alias ci to ci –u.

co (Check out): Pulls information from the RCS ,v file. Files can be checked out unlocked, –u, or locked, –l. The –r option allows any older version to be selected. You can also select revisions by date or user name. In addition, co performs keyword text substitution. If the string $Id: $ is found, it will be replaced with the file's name, revision number, date, author, and locker. It is a good practice to place $Id: $ at the top of each file checked in with RCS. Other keywords are listed in the disk-based documentation.

diff: An excellent implementation of a popular difference generator is included with RCS. The command RCSDiff calls diff for the dirty work. Diff compares two files, showing any differences with > and <. Lines unique to the first file are prefixed with <. Lines unique to the second are prefixed with >. It's easy to keep the arrows straight if you remember "the arrow points at the name on the command line."

ident: Searches a file for keywords of the form $Keyword: $. Ident is sometimes used to extract embedded revision numbers from binaries. All of the RCS binaries contain information in ident format. The AmigaDOS version command performs the same type of function for 2.04 Workbench.

merge, RCSMerge, diff3: Show or process the relationship of two different modifications to the same base file. (More on these commands later.)

RevLabel: This script attaches a symbolic revision to a set of files. Later, you can extract files using this symbol, even if the individual file version numbers vary wildly. It is a good idea to RevLabel each major or customer release.

IT BRANCHES, SINGS, AND DANCES

Imagine you've released versions 1.0 and 1.1 of your product. You are happily working on version 2.0 when an important customer finds a major bug in 1.1. With RCS the situation, while not pleasant, is at least manageable.

RCS can keep track of multiple development threads (see Figure 2). With it, you can make revisions to version 1.1, without duplicating the source code or interfering with development of version 2.0. When the time comes to apply the same fixes to version 2.0, the RCSMerge command automates the work. RCSMerge performs a three-way diff, resolving conflicts between two modifications of the original 1.1 code. Any overlap (lines modified by both the bug fix effort and the 2.0 development) are highlighted for special attention. The accompanying disk contains a sample of such branching.

RCS is not limited to source code. Any sort of text may be enshrined in RCS. Commodore uses RCS for keeping track of books in the library, compact discs available for listening, standards documents that may evolve over time, and public "idea" files to which everyone is encouraged to contribute. Creative uses for RCS abound.

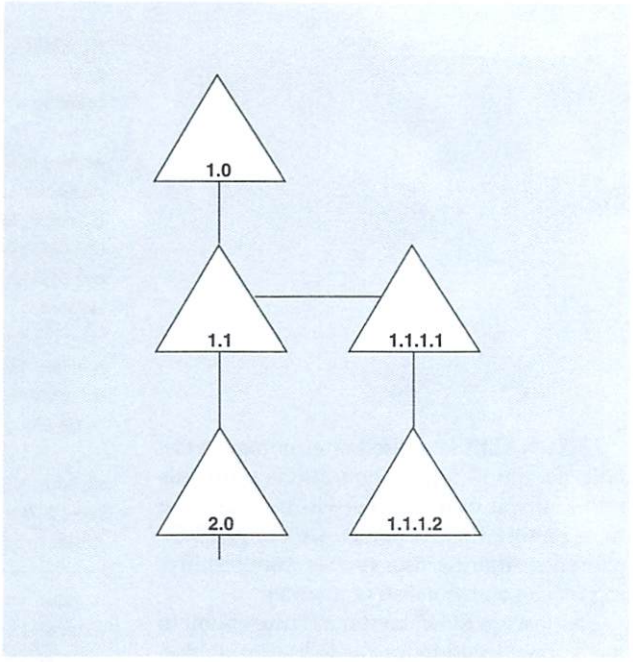


Figure 2. RCS branching helps manage multiple development threads .

On the light side, RCS helped establish the Commodore-Amiga World Record for the longest undetected serious bug. Early in 1991 a ten-instruction timing hole was discovered in Exec PutMsg(). Searching the RCS logs revealed the source of the problem. A certain programmer had optimized the code for space years before. RCS had faithfully recorded the time, date, and changed lines. The date of change? Mid-1985.

RCS has a few more handy tricks for you. For example, to check-out all RCS files for viewing type:

```
spat "co" RCS/#?
```

You can show all files with existing locks via:

```
spat "rlog -L -R" RCS/#?
```

If you happen to forget to lock a file before making changes, don't worry. Simply type:

```
rcs -l filename.c
```

To compare the current working file with the last checked-in version, issue the command:

```
rcsdiff filename
```

Comparing two revisions, on the other hand, requires:

```
rcsdiff -r1.1 -r1.0 filename
```

Go ahead, give RCS a try on the sample files. You'll soon be ready, and eager, to let it help you with your own code. ■

Bryce Nesbitt, formerly a Commodore-Amiga employee, worked on many technical aspects of the Amiga, authored Enforcer, and headed the Operating Systems Development Group for the final phases of the Release 2 project. Contact him c/o The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 02458, or on BIX (bnesbitt).*

From p. 16.

```
   END IF
   LIBRARY CLOSE
END
```

THE C PROGRAM

This program needs to know no Amiga structures, and uses no "standard" C library commands. So, amazingly, it needs no include files at all! We might still choose to include proto/dos.h or libraries/dos.h—these would help with definitions such as that of MODE_OLDFILE.

Our program does not need to open any system libraries. We use functions from dos.library only; and that has been automatically opened by the C system.

To keep the program short, I have not included extra coding to read the contents of the temp file.

```
/* This demo program shows how to */
/* invoke Amiga's DOS library */
/* EXECUTE routine from C coding */

/* We must use Open/Close to allow */
/* operation from Workbench start! */

main()
{
   long handle;
/* Open NIL: as a file to allow output path */
   handle = Open("NIL:",1005); /* MODE_OLDFILE */
   if (handle != 0)
/* Invoke the dos.library Execute function */
/* success = Execute(commandString, input, output handle) */
   {
      Execute("list >RAM:temp lformat=%s", 0, handle);
/* the file handle to NIL: has done its job; close it */
      Close(handle);
   }
/* This example won't show the code to list file RAM:temp */
}
```

EXTRA WORDS

Some varieties of BASIC for the Amiga have keywords that allow commands to be sent directly to the CLI/Shell. For example:

```
ABASIC: SHELL "list >RAM:temp lformat=%s"
COMAL: PASS "list >RAM:temp lformat=%s"
```

Such features are convenient and save you the trouble of using coding similar to that shown above. But my objective in this column is to open the door to reaching the Amiga's inner workings. So even if you know a short cut or another method, check out the approaches given here. ■

Jim Butterfield, a grizzled veteran of the microcomputer wars, has written about Commodore machines from the Kim1 to the Amiga. He is also the author of Machine Language for the Commodore 64, 128, and other Commodore Computers. *Write to him c/o* The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458.*

From p. 20.

even the amount of space you have for presenting the window. You may also want to layout the gadgets again. Because a well-behaved application needs to consider these anyway in the multiple-screen-resolution environment, screen jumping may be as simple as the lines of code above.

FINAL TOUCHES

There only remain a few public-screen routines provided by Intuition. These fall into a class of operations that are most likely to be performed by a public screen manager. One such example was provided on the Amiga Developer Conference disks from Atlanta. There is also a public screen manager called PSX making the bulletin board rounds. Unless you really want to write a public-screen manager, you should avoid these routines:

• **SetDefaultPubScreen(name):** Allows you to make a named screen the default public screen that everything open on. The obvious companion GetDefaultPubScreen(buf) allows the screen manager to inquire which is the default.
• **SetPubScreenModes(modes):** Allows setting some global modes that affect all public screens. Currently the only two modes defined are: SHANGHAI and POPPUBSCREEN. SHANGHAI causes all Workbench-bound windows to be put on the default public screen instead. Note that some applications (such as Workbench) do not open on a screen as a public screen, but instead directly on the screen. The result is that the public screen does not get the window as expected. As more applications use public screens, this problem should diminish. POPPUBSCREEN causes Intuition to automatically pop a public screen to the front whenever a visitor window is opened on it. Lastly, no screen manager would be complete without the ability to list the public screens in the system. LockPubScreenList() returns a pointer to a List structure that can be quickly copied into another location. Note that there is no notification of any changes to the list. Also, as long as the list is locked, you will not be able to open any new public screens, so you should call UnlockPubScreenList() as soon as possible to avoid any deadlock situations.

PUTTING IT ALL TOGETHER

With 2.0, the concept of public screens allows applications to more tightly integrate with one another while at the same time eliminating much of the knowledge of each other. Creating a public screen requires coming up with a name and passing a new tag. Creating a visitor window for that public screen just requires finding a name of a window to open up on. You should make every effort to support jumping at the application level as it gives the user the ability to combine applications in his own way. The end result is more flexibility to the user and a more tightly coupled application platform. ■

John Toebes is the sysop for CompuServe's AmigaTech forum and was a major developer of the SAS/C system for AmigaDOS. Contact him c/o The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458, or on CompuServe (72230,303).*

ON DISK

Better Gadgets with Boopsi

By David "Talin" Joiner

THIS ARTICLE IS not for the faint of heart. We're going to delve deep into the innards of Intuition, and explore vast new realms untouched by the hands of application programmers, into the dark, mysterious realm of... Boopsi.

Boopsi? You may be wondering what a ridiculous name like Boopsi has to do with the Amiga. (I wondered the same thing.) Nevertheless, Boopsi is serious stuff, and stands for the *Basic Object-Oriented Programming System for Intuition*. (For a primer on Boopsi by its author, see "An Introduction to Boopsi," by Jim Mackraz, p. 38, August/September '91.)

WHAT BOOPSI IS (OR ISN'T)

Boopsi is a method for creating objects that can function much like Intuition gadgets, but with more capability. It essentially allows you to "open up" Intuition and add extensions to it. Boopsi objects can be like gadgets or images, or they can act like something entirely new.

Boopsi is not an object-oriented language. It does not interface to an object-oriented language any differently than it would interface to any other language. Boopsi is not itself a language. In fact it is only "object-oriented" if you choose to use it in that fashion.

At the lowest level, Boopsi consists of three things:

1. A set of special Intuition functions for creating and managing Boopsi data structures.
2. A set of hooks inside of the Intuition that can call user code.
3. A set of standards for writing functions that will allow them to communicate with one another.

This article, and the files in the accompanying disk's Joiner drawer, provide a complete and definitive example of a large and powerful Boopsi class. My intention is to provide you with something you can use right away.

THE ULTIMATE SLIDER

I confess, I'm a bit of a gadget nut, and I haven't been satisfied with either the Intuition PropGadgets or the GadTools scrollers. Anybody who has used the GadTools SCROLL_KIND gadgets knows what an improvement they are over the old approach. But, you still can't put them in window borders and you can't make them "stretch" as a window gets bigger. Consequently, I decided to design a new kind of gadget.

I call them "Slider" or "SliderClass" gadgets to distinguish them from PropGadgets. Here are some of their attributes:

• Dragging and rendering occurs in Intuition's task rather than the application task so that dragging appears crisp and smooth regardless of what the application is doing (this rules out "roll-your-own" type gadgets right away). In addition, as the knob drags it won't flicker or flash as it redraws.
• The arrows at the end of the slider are incorporated into the gadget, so the result is one single integrated gadget rather than three gadgets that pretend to be a single gadget, as in GadTools. That way, gadgets can be removed from the gadget list without you knowing anything about it.
• Numbers, or any arbitrary text, can be rendered in the knob of the gadget. For example, an address book program could be designed where the current page letter appears in the gadget's knob.
• Complete control is available over gadget appearance, such as borders, glyphs, patterns, and so on. But, all of these things default to a reasonable appearance so you don't have to mess with them if you don't want to.
• The internal variables of the slider are LONGWORDs, so you're not limited to 64K values. Sliders can start from numbers other than zero, and range to any maximum value.
• They can be put in window borders, and they can change color when the window becomes active, just like the sliders on Workbench.
• GRELWIDTH and GRELHEIGHT flags are used to make the gadget size relative to the window.
• All the parts of the gadget communicate with each other without the application having to do anything.
• Setting up a bunch of different structures to create sliders isn't required. All that's needed is one function call for creation, and one for deletion.
• They look nice and clean, with no pixels over-running the wrong areas.

In addition, Boopsi offers you several other benefits "for free:"

• Gadget attributes can be changed by the application at any time, without having to remove the gadget from the window's gadget list.
• Boopsi gadgets can actually send messages to each other in real time, without the application having to do anything. You can have a slider that tells a list to scroll, for example.

The example in the Joiner drawer on the disk demonstrates the SliderClass, while a demo program exercises a number of its capabilities.

FEATURES OF THE SLIDERCLASS

The SliderClass is implemented as a file called bslider.c, *Here's a "super slider" you can tailor to fit your application.*

along with various support files. Because it is a private class, you must link SliderClass into your programs and initialize it by hand. First, you need to call initSliderClass(), which initializes the class structure and allocates some global data needed by all sliders, and returns a pointer to the newly-created class. (When the program exits, be sure and call freeSliderClass() to free up this data).

To create a slider, call the Intuition function NewObject(), passing to it the class pointer returned by initSliderClass(). You'll also need to pass it a TagList describing the kind of slider you want. The tags that are currently supported are as follows:

GA_Left, GA_Width, GA_Top, GA_Height: Allow you to specify the dimensions of the slider's hit box. You can also use GA_RelRight, GA_RelWidth, GA_RelBottom, and GA_RelHeight to specify a gadget that has dimensions relative to the window's width and height.

GA_Image: Specifies the image class to be used as a frame around the slider. Important note: You must not use a normal Intuition image, because these are fixed in size. You must use a custom image class or Boopsi image class that understands the IM_DRAWFRAME message. Also, if the slider has arrows, this frame will not be drawn around them, only around the actual slider container.

GA_SelectRender: Similar to GA_Image, except that this frame is drawn around the actual container, rather than the slider's hit box. The difference between them is that the container is inset somewhat from (smaller than) the hit box. You can use a combination of GA_Image and GA_SelectRender to get fancy "troughs" for the slider knob to slide into (although I should note that this is contrary to Commodore's *Amiga User Interface Style Guide* (Addison-Wesley); "pushed in" areas are supposed to be read-only).

GA_ID, GA_UserData: Set the Gadget's ID and UserData, which are available for application use.

GA_Previous: Allows you to link gadgets together as you create them. The address of the previous gadget in the chain is passed as the value of this tag item.

GA_DrawInfo: The DrawInfo structure obtained from the screen using the Intuition function GetScreenDrawInfo(). It is employed by the SliderClass to figure out what pen numbers to use when drawing the gadget.

SLD_Vertical: A Boolean value that indicates whether the slider is to be horizontal or vertical. (Note that 2-D sliders like PropGadgets are not supported. My personal feeling is that such 2-D gadgets should be visually distinctive enough to warrant a class of their own.)

SLD_Arrows: A Boolean value indicating whether or not you want arrows. This tag item defaults to TRUE.

SLD_FixBody: Indicates that you want the knob size to be a fixed number of pixels, rather than have it calculated on the fly. The tag value specifies how many pixels you want the knob to be. The default is 0.

SLD_MinVal: The slider's minimum value.

SLD_MaxVal: The slider's maximum value.

SLD_CurVal: The slider's current value. (I've decided to use my own tags here instead of the system-defined ones, such as PGA_TOP, because the meanings of these tags are slightly different.)

SLD_Span: Corresponds to the PropGadget's BODYSIZE variable. It indicates how much of the range (MinVal -> MaxVal) should be represented by the Slider's knob size. For example, if MinVal is zero, and MaxVal is 30, and Span is set to 10, then the knob will appear to be approximately one third of the size of the container.

SLD_FullRange: Normally the slider's value is limited to (MaxVal – Span). For example, if you have a scrolling list with 20 entries, and you can view five of them at a time, then when the slider is all the way at the bottom you should be looking at items 15 through 19. Thus, the slider is limited to values of 0 through 15. When the SLD_FullRange flag is set, however, the limit is raised to MaxVal so that you can set the slider all the way to value 20. This is useful if the slider is being used to represent the value of a dimensionless point. This tag item defaults to FALSE.

SLD_TextHook: Allows you to specify a function call-back for generating the text in the slider's knob. (The default value of NULL means "no text.") SliderClass will pass to you a pointer to the gadget, a structure containing the slider's current values, and a pointer to a buffer to fill in. You fill in the buffer with whatever text you want, and then return the length of the string that you put in the buffer. See the function StdDigits() in cgsupport.c for a better example.

SLD_KnobFont: Allows you to specify the text font that the KnobText (as defined by SLD_TextHook) will be rendered in. The default value of this TagItem is the current mono-spaced system font.

SLD_ArrowSize: This numeric attribute allows you to specify how much of the slider's hit box is taken up by the arrows.

SLD_ContInsetX, SLD_ContInsetY: Allow you to specify the amount that the container rectangle will be inset from that gadget's hit box. Normally, these values are determined by the presence or absence of frame images for the slider and container, and whether or not the slider is horizontal or vertical. If you change any of those attributes, the Inset variables will ►

be set back to their defaults, so if you do decide to have a custom Inset setting, make sure that the SLD_ContInsetX and SLD_ContInsetY come *after* the tags for these other attributes.

SLD_Globals: The image class structures used by the arrow gadgets and the pattern for the container are stored in a structure called SliderGlobals. By default, the sliders will use a standard SliderGlobals created when the SliderClass was initialized. However, by using this tag item, you can substitute your own SliderGlobals structure and make the slider look very different.

ICA_TARGET: As the slider is being dragged, it will broadcast messages indicating it's current value. ICA_TARGET specifies where those messages should be sent. The tag value can be either the address of another Boopsi object, or it can be the special value ICTARGET_IDCMP, which indicates that the message should be sent to the window's UserPort as an IDCMPUPDATE message.

ICA_MAP: As the slider sends messages, it also sends codes that indicate the kind of data that is being sent. These codes are sent in the form of tags. The slider is capable of sending two codes—one that is sent when the slider is dragging (SLD_CurVal) and one that is sent only after the slider has come to rest (SLD_FinalVal). The ICA_MAP tag lets you "remap" these values to any that you want. The actual"map" is a pointer to a tag list. Each element of the tag list is a pair of values—the original and the new, remapped values.

Once the slider has been created, you will need to link it into your window's gadget list using AddGList(), and render it using RefreshGList(). When you are done with the gadget, you can free it from the window's gadget list using RemoveGList(), and free it using the Intuition call DisposeObject().

Getting messages from the slider is simple, just listen for the IDCMP event IDCMPUPDATE. Attached to this message will be a tag list of changes that have occurred. Each attribute will be identified by it's tag value, which in the case of the SliderClass is the same value as specified for the values specified for ICA_MAP.

In the demo code, I have two sliders in the window borders. These sliders have been told that their output attribute codes are PANVAL_X for the horizontal one, and PANVAL_Y for the vertical one. PANVAL_X and PANVAL_Y are arbitrary numbers that I made up; the actual number doesn't matter, they are just codes to determine which slider the value came from. In the Intuition event loop, when an IDCMPUPDATE message is received, we run though the tag list, looking at each value. For each value seen, an appropriate action is taken. In the case of PANVAL_X and PANVAL_Y, we might store the associated slider output value into a variable, and then set a flag indicating that the window's contents should be scrolled. When the "batch" of messages from the window's UserPort is completed, we can look at the flag and scroll the window to the new values if it is set. This allows a nice smooth scrolling action, without SuperBitMap windows.

Another way to get values from the slider is the Intuition GetAttr() function. This allows you to get the slider value at any time, but it is not as efficient.

You can use SetGadgetAttrs() to set the value of the gadget, and you can use it to change other attributes of the gadget as well. There is also a SetAttrs() function. The difference is that SetGadgetAttrs() handles the problems associated with changing an attribute of a gadget that is currently attached to a window, and SetAttrs() is a faster version that doesn't handle that. So if your gadget isn't attached to a window, or it is a Boopsi object that is not a gadget type, you'll want to use SetAttrs() instead.

GENERAL CODE CONVENTIONS

Before I get into the details of the SliderClass, I should explain some of the code conventions I used.

A handy little structure that is defined in intuition.h and is used to represent a rectangular region is struct IBox:

```
struct IBox {
   WORD  Left;
   WORD  Top;
   WORD  Width;
   WORD  Height;
};
```

I use this structure everywhere. In fact, in my personal library I have a bunch of functions that operate on IBoxes, such as intersect, shrink, and so forth.

Another structure that I use a lot is struct ValueRange, defined in newgadgets.h :

```
struct ValueRange {
LONG        Min,
            Max,
            Current,
            Span;
};
```

I use a ValueRange structure to represent the current slider values, and I use them for other types of gadget classes as well.

In this article, I'm using the term "message" fairly generically to mean a structure that is passed to a function and that has a code in the structure that tells the function what kind of structure it is. An example of this is the opUpdate structure found in intuition/classusr.h:

```
struct opUpdate {
ULONG               MethodID;        /* Message type */
struct TagItem      *opu_AttrList;   /* TagList of attributes to update */
struct GadgetInfo   *opu_GInfo;      /* GadgetInfo structure used to
                                        rerender */
ULONG               opu_Flags;       /* various flags */
};
```

This structure is used to send an OM_UPDATE or OM_NOTIFY message to a Boopsi object. The first field, MethodID, would contain either the value OM_UPDATE or OM_NOTIFY.

Note that in some cases, these structures are actually built on the stack out of parameters. For example, to send an OM_UPDATE message to an object, you don't need to actually fill in an opUpdate structure; you can build it like this:

```
result = DoMethod(object, OM_UPDATE, taglist, gadget_info, flags );
```

Note that each of these parameter values is pushed on the stack. If you take the address of the second parameter (OM_UPDATE), you essentially have an opUpdate structure! This is exactly what DoMethod() does when it sends a message to your object.

This technique only works because function calls on the Amiga normally push parameters as four-byte values, and the opUpdate structure has been designed only to use four-byte values.

There is also another version of DoMethod(), called DM(), that doesn't do it this way. Instead it expects a filled-in opUpdate structure (or whatever structure you may be sending) as it's second parameter. In addition to DM and it's associated functions (all of which are taken directly from the original Boopsi example code written by Jim Mackraz), I have my own version of DM(), called SendObject(), which takes advantage of registerized parameter calls. Fortunately, these functions are so tiny it doesn't hurt to have both versions linked in.

IMPLEMENTATION OF THE SLIDERCLASS

The first question to consider when creating a Boopsi class is: "what will this class's superclass be?" Under 2.0, each of the standard Intuition gadgets is implemented as a gadget class, and a customized Boopsi class should be written as a subclass of one of the existing classes. Each time an event is sent to a gadget, the gadget's class can decide to let the superclass handle the event instead. This can allow for reusing a lot of code.

For example, in the case of the SliderClass, it would seem that because a slider is a lot like a standard Intuition PropGadget, it would make sense to make Slider a subclass of PropGadget. Upon closer examination, however, I realized that the internals of my SliderClass had almost nothing in common with PropGadgets, and, in fact, several of PropGadget's features would actually get in my way. So I chose instead to make SliderClass a subclass of gadgetclass, which is the parent class of propgadgetclass. Figure 1 shows how the SliderClass fits into the Boopsi class hierarchy.

The code for the SliderClass consists of essentially four parts. These are described in the following paragraphs.

RENDERING AND CALCULATING CODE

The first part is a set of functions that draws the various parts of the slider, and also determines where to place the knob and how big the knob should be. The main functions are as follows:

QuickBevel(): Draws a beveled box. You'll note that there are other functions for drawing beveled boxes as well, this one is just very simple and fast. We don't want the slider to hog the CPU while being dragged with the mouse.

RenderSlider(): Draws all the components of the slider, such as the container, the border frame, the knob, the arrows, and so on. You set flags indicating which parts you want drawn, because when dragging the knob with the mouse it isn't necessary to rerender the arrows or the border frame. This function also calculates what color the gadget should be, based on whether it and/or the window is active. To avoid flicker, it is careful to never draw any pixel more than once.

CalcSliderBody(): Calculates the size of the slider knob, given such factors as the span of the slider, the size of the gadget, various flags, and so on. It also ensures that the knob will be large enough to grab, and if there is text inside the knob, it figures out how big that text should be and adjusts the knob's size accordingly.

CalcSliderPosition(): Once the size of the knob is determined, this function figures out where the knob should be placed. Various flags and conditions can affect this calculation, such as the size of the arrow gadgets, whether the slider is vertical or horizontal, and so on.

SetUpContainerInfo(): When the slider is rendered or selected, this function is called to figure out the size of the gadget and the size of the knob's container. Note that in the current implementation, the code assumes that the window (and thus the gadget) will not change size while the slider is actually being dragged. ►

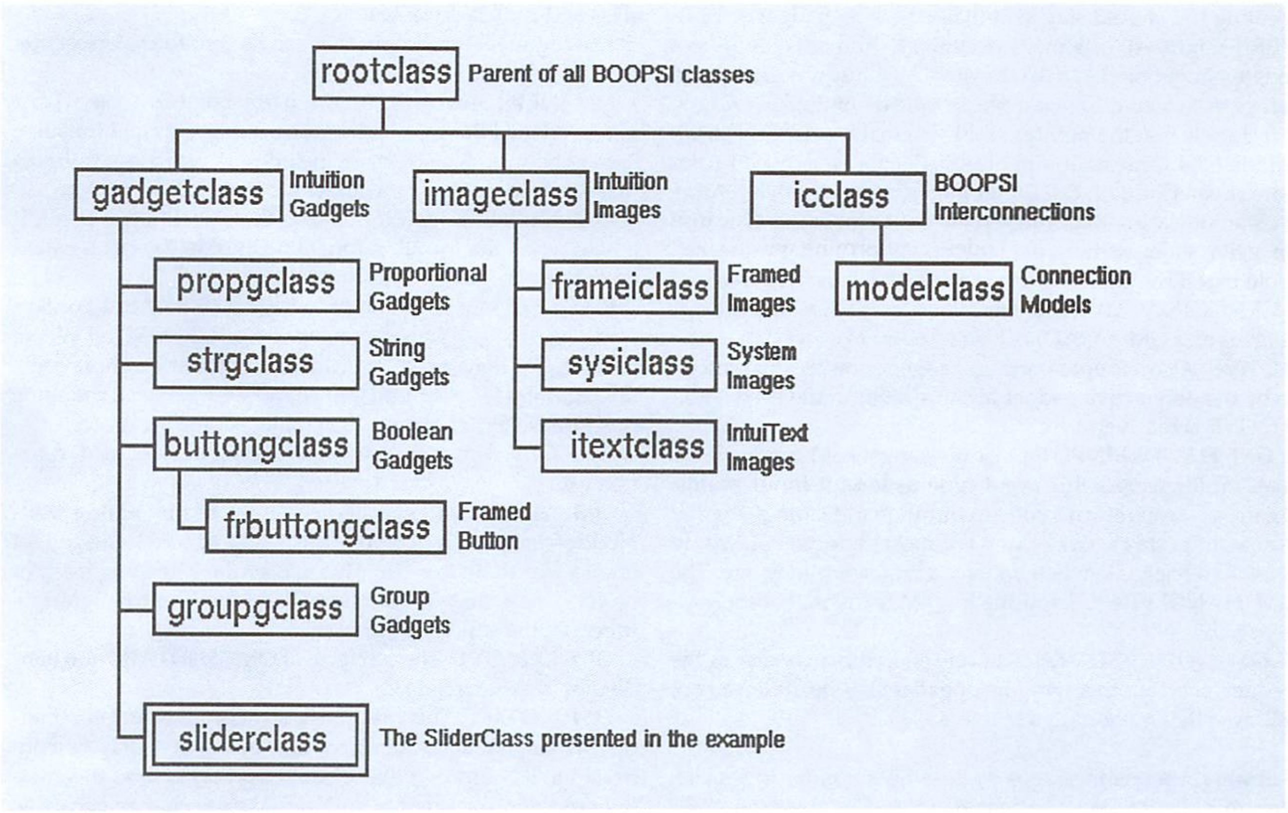


Figure 1. The Boopsi class hierarchy. Note the position of SliderClass.

SetUpKnobInfo(): This function fills in a KnobInfo structure, which is used to hold all the temporary variables for calculating knob position and size.

TestArrow(): This function tests to see if the mouse pointer is over one of the arrows. If it is, it returns the part code for that arrow.

LOW-LEVEL EVENT HANDLER

The low-level event-handler section of code handles input events passed to it by Intuition. These events originate in the input.device task, which calls Intuition's input handler. Intuition converts the input events to gadget events and passes them to the handler for the currently active gadget. The gadget-handler code (our code, in the case of the SliderClass) is free to interpret these events however it wishes. Note that all of this occurs in the context of the input.device task, not the application task, so special care is necessary, especially when rendering.

There are basically five different message types that need to be supported:

1. **GM_HITTEST:** The gadget code should return TRUE if it was actually hit. Note that this function should be used only to decide whether the gadget was actually "touched" by the mouse, and should not be used to decide whether to go active or not. That decision should be made by the GM_GOACTIVE handler. Otherwise, gadgets that are "underneath" the gadget (for example, a gadget that is overlapping a window drag bar) might be activated "through" your gadget.
2. **GM_RENDER:** This is a command from Intuition to rerender your gadget.
3. **GM_GOACTIVE:** This is Intuition's way of letting the gadget know that someone has clicked on it, and that it should go to an active state. The gadget can return codes indicating if it should stay active (like a string gadget or RELVERIFY button), or should de-activate immediately (like a toggle gadget or GADGETIMMEDIATE button). It can also refuse to go active; for example, in the case of the SliderClass, if it detects that the pointer to the original InputEvent structure is NULL (meaning that the gadget was activated from an ActivateGadget() call rather than a user action), it returns a code indicating that it doesn't want to go active (because an active slider without user manipulation makes no sense). Note that if the gadget does go active, it is guaranteed to get a GM_GOINACTIVE call eventually, so you can allocate resources on a GM_GOACTIVE and free them on a GM_GOINACTIVE. Also, except for group gadgets, you are guaranteed to be the only active gadget in the system until GM_GOINACTIVE is received.
4. **GM_HANDLEINPUT:** Once the gadget has been made active, Intuition uses this event type to send it input events. Again, you can return a code to Intuition indicating whether you want to stay active or not. For most gadgets, you would want to de-activate when you get a mouse-up message. The GM_HANDLEINPUT routine is what really makes the gadget go.
5. **GM_GOINACTIVE:** Intuition sends this message to the gadget when another window or gadget has been clicked on, telling it that is should go inactive.

Intuition sends messages by passing a pointer to a structure to your low-level gadget handler. The message types listed above are passed in one of the structure's fields. Note that some of the fields in the last part of this structure may be different for various messages. All of the structures are defined in the include file intuition/gadgetclass.h.

The codes returned by GM_GOACTIVE and GM_HANDLEINPUT are as follows:

GMR_MEACTIVE: Return this if you want to go active (or stay active if you already are).

GMR_REUSE: Return this if you want to go inactive, and you want Intuition to reuse the input event. For example, gadgets should go inactive whenever the right mouse button is pressed, so that menu activity can occur. You would return GMR_REUSE to instruct Intuition to reuse the mouse button event for menu-processing.

GMR_NOREUSE: Return this if you want Intuition not to reuse the event. A typical case would be a mouse-up message.

In addition, you can set the GMR_VERIFY flag in either the GMR_REUSE or GMR_NOREUSE codes. This instructs Intuition to send a GADGETUP message to the UserPort. You can use this feature to implement a RELVERIFY type of Boolean gadget.

HIGH-LEVEL MESSAGE HANDLER

The high-level message handler monitors messages from sources other than Intuition, for example, the application or other Boopsi objects. Unlike the low-level handler, high-level messages are really just function calls that can occur in either the Intuition task or the application task.

High-level messages can be "inherited" from higher-level classes in the Boopsi inheritance tree. We can see the usefulness of this just by examining the tag values that the SliderClass accepts. Only the tag items that begin with SLD are actually implemented by theSliderClass. All the others, such as GA_Left and ICA_TARGET are inherited from the gadgetclass and rootclass classes.

The high-level messages that can be sent to the object are:

OM_NEW: Sent by the Intuition function NewObject() to a newly-created Boopsi object, telling it that it should initialize itself. The message structure includes a tag list of attributes that can be used to set the initial attributes of the gadget.

OM_DISPOSE: A command to the object to free itself. It is sent when the Intuition function DisposeObject() is called on the object.

OM_SET: Sent to the object whenever a SetAttrs() or SetGadgetAttrs() call is made on the object. Like OM_NEW, the message contains a tag list of attributes. In the example code, the handlers for both OM_NEW and OM_SET call the same function, setSliderAttrs(), to actually set the attributes.

OM_GET: Sent to the gadget as a result of a GetAttr() function call.

OM_ADDTAIL: Each Boopsi object begins with a MinNode structure. You can use this message to tell the gadget to add that node to a list. This is useful for keeping track of objects. Note that in the SliderClass, we don't handle this function, the superclass does.

OM_REMOVE: The converse of OM_ADDTAIL, also handled by the superclass.

OM_NOTIFY: This message instructs the object to broadcast its current state to any other objects that may be connected to its outputs. Objects are supposed to send this message to themselves when an internal state change occurs as the result of some low-level event. For example, in the Slider-

Class, we send an OM_NOTIFY to ourself whenever the slider's value changes. Why not just broadcast the data directly? Because the OM_NOTIFY function isn't handled by the SliderClass, but rather by the superclass, gadgetclass. When we send an OM_NOTIFY to ourselves, along with a tag list of changed attributes, it actually gets passed to the OM_NOTIFY handling function in the gadgetclass class, which then takes care of distributing the message to the correct receivers. Each receiver gets to look at the tag list and can take appropriate action. In addition, if someone writes a subclass of SliderClass, that class will be able to intercept and veto or modify the notification message by overriding OM_NOTIFY in their class.

OM_UPDATE: This message is sent to the object by other Boopsi objects. Its use is to allow objects to interconnect with each other. One Boopsi object can send an OM_UPDATE message to another one, telling it to update its internal fields. Interconnection, or "IC" objects can be built that translate the output of one object to a set of codes understandable by another object. In my code, OM_UPDATE and OM_SET do essentially the same thing.

OM_ADDMEMBER, OM_REMMEMBER: Boopsi allows the concept of group gadgets, in other words groups of gadgets that are tightly coupled with each other. One example might be a "mutual exclusion group." This would function as a group of mutually excluding gadgets. You could use OM_ADDMEMBER to add gadgets to the group, and the handler code for the group could ensure that no two gadgets are selected at the same time. (Note: There are actually about five different ways to implement mutual exclusion in Boopsi!)

One thing to note about OM_SET and OM_UPDATE is that they need to keep track of whether the state of the gadget changed, so that they can rerender the gadget if necessary. If the superclass makes any changes, these must be kept track of as well. The way to do this is as follows: Each OM_UPDATE or OM_SET handler function should compute and return a changed flag, which is TRUE if any changes were made that might cause a refresh. When the handler calls its superclass, the superclass will also return a changed flag, which should be combined via OR into the flag for this handler as well. Finally, if this handler is found at the bottom of the tree (by checking to see if the objects "true class" is the same as the class being handled by the handler function), and there were changes, then the handler function should cause a rerender. Only the bottom-most method handler should cause a rerender, otherwise the gadget would be redrawn multiple times.

SUPPORT FUNCTIONS

The fourth section of code is the support functions, including the standard Boopsi functions for sending messages to objects and classes, as well as some useful gadget-related functions and miscellaneous routines. Some of these bear further explanation.

Most Boopsi example code uses a "switch" statement to interpret the various message codes. Just to be different, however, I implemented a data-driven message dispatcher. This small assembly language routine takes the message code (called the "Method ID") and uses it to look up the address of the function to call in a table. (Note that Method IDs aren't always contiguous, so there may be several tables.) If the function isn't in the table, it just jumps to the superclass. The result can be much faster than a switch statement, depending on the messages sent. Even faster techniques are possible, using caching.

The C source file cgsupport.c has a bunch of useful func- ►

Level	Message	Meaning	Data
low	GM_HITTEST	Test if gadget was hit by mouse	Mouse Coords
level	GM_RENDER	ReDraw all or part of the gadget	GREDRAW_REDRAW = draw whole gadget
msgs			GREDRAW_UPDATE = draw part of gadget
			GREDRAW_TOGGLE = highlight only
	GM_GOACTIVE	Gadget Activation	InputEvent (or NULL)
	GM_HANDLEINPUT	Handle Input Event	InputEvent
	GM_GOINACTIVE	Gadget DeActivation	
high	OM_NEW	Initialize an object	TagList of attributes
level	OM_DISPOSE	Free an object	
msgs	OM_SET	Set Object Attributes	TagList of attributes
	OM_GET	Query Attribute of object	Attribute and buffer pointer
	OM_ADDTAIL	Add Object to list	List pointer
	OM_REMOVE	Remove object from list	
	OM_NOTIFY	Instruct object to broadcast its current state	TagList of attributes to broadcast
	OM_UPDATE	Receive broadcast from another object	TagList of attributes sent.
	OM_ADDMEMBER	Add item to internal list	Item to add
	OM_REMMEMBER	Remove item from internal list	Item to remove

Figure 2. Standard Boopsi messages.

tions for implementing custom gadgets. It contains a function to invoke image classes (DrawCustomFrame()), a function to calculate gadget hit boxes (SetupIBox()), functions to help objects send messages to each other and to the application (notifyAttrChanges() and updateAttrChanges()), and more.

The following are several additional small assembly language files. These are generally useful functions that I have written or collected over the years:

clamp(): Constrains a value between minimum and maximum limits.

GetHead(): Returns the first node of an Exec list.

NextNode(): Returns the next node of an Exec list.

DigitCount(): Quickly calculates the number of decimal digits in a binary number, without taking the time to convert that number to decimal.

fs2a() ("Fixed Signed to ASCII"): Quickly converts a binary number to ASCII decimal, with a fixed number of digits.

DrawRect(): Quickly draws a hollow rectangle by pushing coordinates on the stack and then calling PolyDraw().

Note that you don't have to implement all of this at once. My advice is to worry about the high-level message section last. First, design your structures and write a function that just draws the gadget. Call this function directly from your test application, so that you can debug it. Remember, once you hook it up to Intuition, you won't be able to use a debugger or printf() calls (though kprintf() will work) because if you halt Intuition, the debugger won't run either! Also, make sure all of your pixels are in the right place. Then work on the low-level message portion, and just stuff in default values when the object is created. Hard-code as many things as you can, but bear in mind you will want to make them variables eventually. Later, you can hook up routines to allow finer control of the gadget by the application using the high-level interface.

Another thing to note is that some of the fields of the original Gadget structure are also available for your use. In SliderClass, the following fields are used:

GadgetRender: Pointer to slider frame image class.

SelectRender: Pointer to container frame image class.

GadgetText: Pointer to knob font.

SpecialInfo: Pointer to a SliderInfo structure which contains additional variables. Note that the SliderInfo structure actually follows the Gadget structure in memory. This is taken care of automatically by the class rootclass when the object is created. For convenience, however, I also use the SpecialInfo pointer to point to the slider data.

These are really about the only fields that are truly safe. Note that you must *not* use the MutualExclude field! This is used by Boopsi to hold a pointer to your class. NextGadget and other fields that are used directly by Intuition probably shouldn't be messed with either.

IMAGE CLASSES

Another module that is not actually part of the SliderClass but is required by the SliderClass in order to work is BoxImage. This is a Boopsi image class for drawing beveled boxes.

BoxImage can draw a beveled or single-color frame around any rectangle that is passed to it, and can also draw a glyph centered within the frame, such as an arrow or icon. A BoxImage structure (defined in newgadgets.h) is used to define the type of frame to be drawn and the glyph. The PlanePick and PlaneOnOff fields are used to define the Width and Height of the glyph, respectively.

Image classes can be called either by the application, by Intuition, or by a gadget class. There are a variety of messages that can be sent to the image class, such as IM_DRAW and IM_DRAWFRAME. The only difference between the two is that IM_DRAW specifies that the image should use the width and height variables in the actual image structure, whereas the IM_DRAWFRAME message includes the width and height to be used in the message structure itself. This means that a single image class structure can be used by many gadgets, regardless of their size. Note that Intuition only uses the IM_DRAW message, so you won't be able to do this trick with normal Intuition gadgets.

The function DrawCustomFrame() in cgsuppport.c can be used to invoke image classes. You pass it a pointer to a RastPort, a pointer to the image, an IBox which contains the rectangle specifying the rectangular region to draw the frame in, a DrawInfo which represents the pen colors for the screen, and finally the "state" that you want the image to be drawn in, such as "selected," "ghosted," and so on. Note that the pointer to the image must point to the image field in the BoxImage structure, *not* the beginning of the structure. This is because there is a hidden object structure that comes before the image. The DrawInfo is used to make sure that the bevel-boxes render in the correct colors, even if the screen has a nonstandard palette. Note also that you can link several image classes together using the NextImage field, just like normal Intuition Images. Also, the pointer must point to the Image structure, not to the beginning of the BoxImage structure.

The SliderClass rendering code calls DrawCustomFrame() for each of the arrow gadgets, using the image class pointers specified in the current SliderGlobals. These structures have the little arrow glyphs already set up by the class, however, you can easily replace them with your own.

ABOUT THE DEMO

The demo program demonstrates a couple of other noteworthy things besides Boopsi. For one thing, it can handle any number of open windows and will only exit when there are no windows left. (Use the New menu item to open a new copy of the window.) It uses a number of advanced techniques for handling:

• Intuition messages
• menus
• simple-refresh windows and message ports that would be useful when writing a large, complex application that supports any number of open documents
• and any number of associated windows, open for each document, that share the same UserPort, and thereby have only a single task to deal with.

The demo was made by paring down a large application. Therefore, it might make a good skeleton for future large applications. ■

David "Talin" Joiner is the author of Music-X and Faery Tale Adventure, plus an artist, award-winning costume designer, and moderator of the user.interface topic of the Amiga.sw BIX conference. Contact him c/o The AmigaWorld Tech Journal, *80 Elm St. Peterborough, NH 03458, or on BIX (talin).*

Graphics Handler

From p. 23

Env(). You may also safely pass in NULL. Using the function might take the following form:

```
FreeSprRendEnv (rendenv);
```

SIDE NOTES

You might notice that Amiga sprites are interleaved bitmaps. Thus, interleaved BitMap structures can be created to describe sprites, and interleaved blits can be performed directly into them. If you've read "Blitter Optimization" (p. 10, November/December '91), you're aware of the benefits of this kind of blit, as well as the pitfalls. If you are unfamiliar with these traps, the primary rule to remember is to never render outside the valid area. Remember that the BitMap is describing a larger area than is truly valid. In particular, calling SetRast() on a sprite's RastPort is unsafe (unless a Layer has been attached), because SetRast() tries to write to all the bits described in the BitMap.

The routine GetSprRendEnv() currently does not handle 16-color sprites, but could easily be extended to support them. The operations are precisely the same, except that plane pointers two and three are filled in with pointers into the second sprite. This might appear as follows:

```
struct SimpleSprite *spr1, *spr2;
struct BitMap  *spritebm;
PLANEPTR  spritedata;


InitBitMap (spritebm, 4, 32, spr1->height);
spritedata = (PLANEPTR) spr1->posctldata;
spritebm->Plane[0] = spritedata + 4;
spritebm->Plane[1] = spritedata + 6;
spritedata = (PLANEPTR) spr2->posctldata;
spritebm->Plane[2] = spritedata + 4;
spritebm->Plane[3] = spritedata + 6;
```

WINDING DOWN

The book of the month is *The Wizardry Cursed* by Rick Cook. My son, Alex, and I have both read it and think it's marv... Oh! Sorry, wrong column.

Rendering into sprites has a number of advantages. For example, sprites could be used to display textual information cleanly on a HAM screen. Sprites can live on top of double-buffered displays without additional programming, enabling you to display limited information while an animation is in progress. They can also be used to peacefully display global information on top of all screens, such as the current time of day, total available memory, the open count on a shared library, or something else of interest. Hopefully, these techniques will add to the tools at your command to help you create exactly the application you want. ■

Leo L. Schwab was the principal programmer behind Disney Presents... The Animation Studio and has created many PD screen hacks. He can frequently be seen at computer shows wearing a cape and terrorizing IBM reps. Contact him c/o The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458, or on BIX (ewhac), Portal (ewhac), or usenet (ewhac@well.sf.ca.us).*

Display Refreshing

From p. 28

namically allocate CallNodes—the LocalData might be a pointer back to the CallNode or to some other data structure associated with this particular CallNode.

There are several other functions provided in the library that add additional functionality to the system and make it useful for applications other than refreshing.

The function RemCallNode() is exactly the opposite of AddCallNode()—it removes a CallNode from a CallList without actually calling the function. Simply call it as you would Exec's Remove() function. As you can AddCallNode(), you can safely call this function with a CallNode that was not previously on any list. (See the source code on disk for the actual definition of this function.)

Finally, if you want to call all of the CallNodes in a CallList, but leave the list intact so you can later use the same list again, just use the Call() function. In this case, an explicit call to RemCallNode() is required to remove a CallNode. This will generally not be useful for refreshing, but may be useful in other applications of this system.

If you use this variation, you should not call AddCallNode() from within the CallFunctions, because newly added nodes may be accidentally skipped (even if they have a lower priority than the current node). You should also not call RemCallNode() on *other* nodes in the same list from within a CallFunction. However, you may RemCallNode() the *current* node from within its own function, to indicate that that function doesn't need to be called anymore.

There are also two variations of Call() and CallRem(), named CallExt() and CallRemExt(), respectively. These variations accept the same arguments as the regular versions, plus an extra parameter: a pointer to a function to call after each CallFunction on the list is called. You can use this facility, for example, to check for such signals as CTRL-C while the list is being processed. See the source code on disk for more details about these functions.

By now, you have probably noticed that CallLists can be used for applications other than display refreshing. For example, during your program you can maintain a CallList that is called only once, when your program is about to exit. When a module in your program allocates some memory or opens a window, it simply adds a CallNode to this global CallList. When the program is about to exit, this CallNode will automatically be called, so the module can free anything it previously allocated.

Using CallLists can help make screen refreshing elegant and efficient. Unnecessary processing is eliminated and interdependencies can be handled easily. Because the CallList library functions use no global variables, they can be reused as much as necessary in one program, and will cause no problems (if used correctly) in re-entrant code such as run-time libraries. Remember, CallLists are useful for many purposes, display refreshing is just one of them. ■

Bryan Ford is a student at the University of Utah and works on freelance programming projects for local companies. Contact him c/o The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458, or on Internet (bryan.ford@m.cc.utah.edu).*

LETTERS

Flames, suggestions, and cheers from readers.

A FAILED LAUNCH

After reading Marvin Weinstein's "Extending ARexx" (October '91, p. 18), I tried modifying fastmenu to launch SID and Uedit, which do not reliably launch from AmiDock. So I wrote the following program:

```
/**
*
**/
....
call AddGadget(ClickList,6,28,EW_3.0,"
EasyWriter_3.0 ", ,
   "'address command run TOOLS:Ue_3.0 -
   ds:Data!_3.0'"
call AddGadget(ClickList,6,41,WP," Word-
Perfect 4.1 ", ,
   "'address command run TOOLS:Word-
   Perfect/wp'"
...
```

I added the WordPerfect gadget because WP launches reliably from AmiDock. However, I was surprised to find that, just like SID and Uedit, WP does not launch at all. Did I misunderstand something? The logic of this is so straightforward.

Robert A. Jenkins, Ph.D.
Miller Woods, Illinois

"The problem lies not in the logic Dr. Jenkins but in the quoting." This point was covered implicitly in "ARexx Arcana, Hosts and Quotes" (August/September '91, p. 2), but merits further explanation. If we look at the call to AddGadget() used in your ClickList program we see that the string:

```
"'address command run TOOLS:Word-
Perfect/wp '"
```

is parsed once by ARexx. At this time ARexx removes the first set of quotes and notifies your rexxarplib host to send the string program:

```
'address command run TOOLS:Word-
Perfect/wp '
```

to ARexx for processing. When ARexx gets this string it parses off the single quotes (because they tells it that this is a string program) and then attempts to interpret the line:

```
address command run TOOLS:Word-
Perfect/wp
```

Here things break for a variety of reasons. First, ARexx treats a line which begins with a TOOLS:... as a label, which this line is not. Next, it finds two variables, WordPerfect (which it automatically uppercases to WORDPERFECT) and wp (which it converts to WP). It then, thanks to the /, attempts to divide these undefined variables, producing an error message. (Try this line out in an ARexx program to be run from a CLI and see the error message that results.) The same problem appears in all of your calls to AddGadget. To get something that works you would type:

```
address command run "TOOLS:Word-
Perfect/wp"
```

because the quotes would keep ARexx from interpreting the quoted material before passing the command to AmigaDOS. You must use more quotes in your call to get this string passed to your host. The correct form of the call is:

```
call AddGadget(ClickList,6,41,WP," Word-
Perfect 4.1", ,
   "'address command run ""TOOLS:Word-
   Perfect/wp""' "
```

In this case the outermost double quotes are parsed off when the message is sent to the rexxarplib host and, at the same time, the innermost "" are converted to " . When the string program is sent to ARexx the outermost single quotes disappear and ARexx attempts to interpret a viable one-line program.

It is also safer, if you are going to use funny gadget names such as UE_3.0 to enclose them in quotes inside the call to AddGadget() to avoid similar problems.

Marvin Weinstein

JOYSTICK RESPONSE

This letter is in response to Tony Gore's letter ("A Challenge," November/December '91). A joystick may have three buttons, just like a mouse. In fact, the computer cannot tell a joystick from a mouse except that a joystick cannot generate the proper quadrature signals to increment the mouse counters. Secondly, there are three-button joysticks available for the Amiga: Go to a video game retailer and buy a joystick for the Sega Genesis. (I recommend the Sega Genesis Arcade PowerStick.) These joysticks are completely Amiga compatible and have three buttons. Finally, there are some software developers who already write software for joysticks with more than one button. If you try out R-TYPE with a Genesis joystick, you will find that the program uses two of the three buttons (one to fire and one to separate the expansion from the ship).

I agree that more software needs to support multiple fire buttons, but a new joystick standard is not the answer. Joystick manufacturers just need to quit making single-button joysticks and use the joystick specification more fully, as did Sega.

Joseph Fenton
Barker, Texas

WANTED ADA

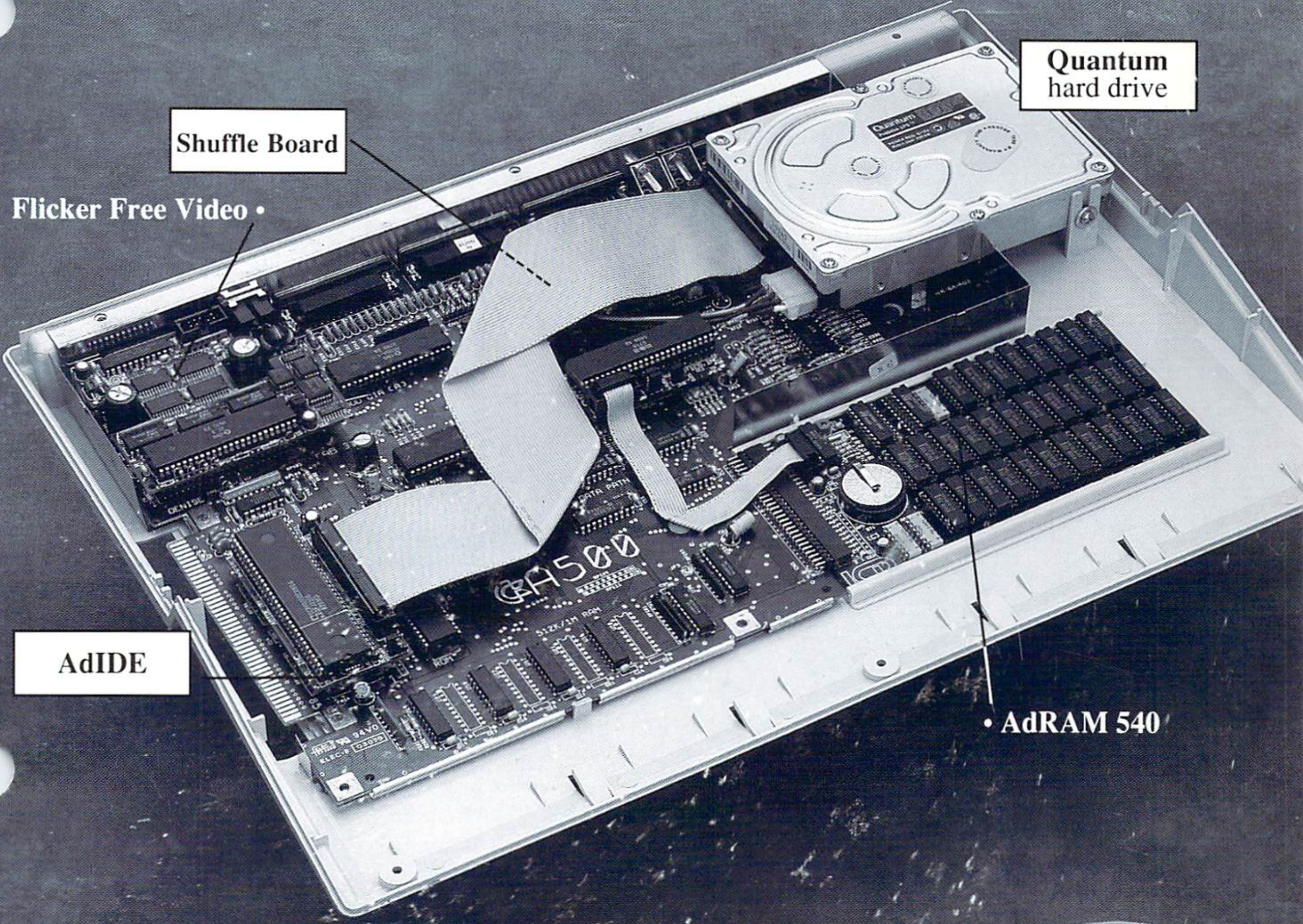
I am studying mathematics and computer science at a college in Sweden. In computer science classes we are using Ada. I think Ada is a good programming language, and I've heard it is making good progress. I encourage developers to develop an Ada programming environment for the Amiga. But hurry! In two years I will start to develop one myself.

Patrik Johansson
Alvesta, Sweden

LET US KNOW

What are your suggestions, complaints, and hints for the magazine, Amiga developers, and your fellow readers? Tell us about them by writing to Letters to the Editor, The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458, or posting messages in the AW.Tech Journal conference on BIX. Letters and messages may be edited for space and clarity.* ■

Prima!
A Look Inside the Ultimate A500.



ICD proudly presents **Prima**™... the high performance, low cost hard drive for Amiga® 500 computers. Prima blends a large capacity, low power Quantum™ hard drive with the **AdIDE**™ host adapter for an unbeatable combination.

Prima replaces the internal floppy drive but includes **Shuffle Board**™ to make your external floppy drive DF0:. **Prima** features auto–booting from FastFileSystem partitions, high speed caching, auto–configuring, and A–MaxII™ support. Formatted capacities of 52 and 105 megabytes are currently available.

Prima comes complete with instructions, software, and all the hardware necessary for a simple, clean, no–solder installation. It does require an A500 with switching power supply, 1 megabyte of RAM, and an external floppy drive for setup and installation.

What other products would we include in the "Ultimate A500"? Of course a four megabyte **AdRAM**™ **540** and **Flicker Free Video**™ with a multi–sync monitor. Why settle for less?

ICD

ICD, Incorporated
1220 Rock Street
Rockford, Illinois 61101
USA (815) 968-2228 Phone (800) 373-7700 Orders (815) 968-6888 FAX

TAKE THIS JOB AND LOVE IT!

The fastest growing video technology company in the world is looking for technical support specialists. We're looking for the kind of person that enjoys solving problems. If you have the unique ability to; hear a problem, ask the right questions, analyze the answers, and then provide a perfect solution, you belong with NewTek. You'll be a key member of the best support team in the business. Best of all, you'll be working for a company that has an intense commitment to serving its customers.

We're looking for rare individuals with as many of the following qualities as possible:

- Excellent written and verbal communication skills
- Experience in video production and editing
- Knowledge of 3D animation and computer graphics
- Knowledge of Amiga software and peripherals
- Experience with Digi-View and Digi-Paint
- Experience with the Video Toaster
- Work experience in the technical support field

The Video Toaster is changing the way the world thinks about video production. The Toaster is being used everywhere from networks to cable stations and our users range from rock stars to wedding videographers. Helping producers, artists, and video makers use this powerful tool to revolutionize video is an exciting and rewarding career. If you're ready to make a change for the better call Kiki Stockhammer at 913-354-1146.

215 S.E. Eighth St.
Topeka, KS 66603
913-354-1146
FAX: 913-354-1584

NEWTEK
INCORPORATED